(2)

ANNUAL REPORT

VOLUME 1
PART 1

TASK 1: DIGITAL EMULATION TECHNOLOGY LABORATORY

REPORT NO. AR-0142-91-001

September 27, 1991

DIGITAL EMULATION TECHNOLOGY LABORATORY

Contract No. DASG60-89-C-0142

Sponsored By

The United States Army Strategic Defense Command

COMPUTER ENGINEERING RESEARCH LABORATORY

Georgia Institute of Technology

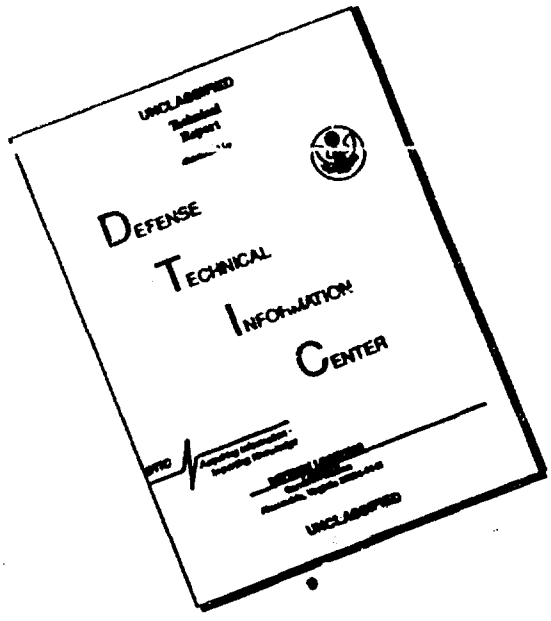
Atlanta, Georgia 30332 - 0540

Contract Data Requirements List Item A005

Period Covered: FY 91

Type Report: Annual

DISCLAIMER NOTICE



THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

REPORT DOCUMENTATION PAGE

Form Approved OMB No 0704 0188

1a REPORT SECURITY CLASSIFICATION: Unclassified

1b RESTRICTIVE MARKINGS:

2a SECURITY CLASSIFICATION AUTHORITY:

3 DISTRIBUTION/AVAILABILITY OF REPORT: 1)Approved for public release; distribution is unlimited
2) continued on reverse side

2b DECLASSIFICATION/DOWNGRADING SCHEDULE:

4 PERFORMING ORGANIZATION REPORT NUMBER(S): AR-0142-91-002

5 MONITORING ORGANIZATION REPORT NUMBER(S):

6a NAME OF PERFORMING ORGANIZATION: School of Electrical Eng. Georgia Tech

6b OFFICE SYMBOL (If applicable):

7a NAME OF MONITORING ORGANIZATION: U.S. Army Strategic Defense Command

6c ADDRESS (City, State, and ZIP Code): Atlanta, Georgia 30332

7b ADDRESS (City, State, and ZIP Code): P.O. Box 1500
Huntsville, AL 35807-3801

8a NAME OF FUNDING/SPONSORING ORGANIZATION:

8b OFFICE SYMBOL (If applicable):

9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER: DASG60-89-C-0142

8c ADDRESS (City, State, and ZIP Code):

10 SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO	WORK UNIT ACCESSION NO

11 TITLE (Include Security Classification): Guidance, Navigation and Control Digital Emulation Technology Laboratory Volume I (Unclassified) Part 1, 2 and 3

12 PERSONAL AUTHOR(S): C. O. Alford, Thomas R. Collins, Stephen R. Wachtel

13a TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
Annual	FROM 9/28/90 TO 9/27/91	9/27/91	434

16 SUPPLEMENTARY NOTATION:

17 COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

Part I

1. Introduction
   - 1.1 Objectives
   - 1.2 Schedules & Milestones
2. Hardware & Facilities
   - 2.1 Parallel Function Processor
   - 2.2 Seeker Scene Emulator
   - 2.3 Other computer systems
   - 2.4 Secure Laboratory
3. FPP/FPX Development Tools
   - 3.1 Introduction
   - 3.2 FPP/FPX object module loader
   - 3.3 FPP/FPX program downloader
4. Software Development Tools
   - 4.1 Introduction
   - 4.2 Sequential Programming Tools
   - 4.3 Parallel Programming To
   - 4.4 Sepcial purpose tools
5. Application Software
   - 5.1 EXOSIM
   - 5.2 LEAP
6. Appendix A: Environment Fil Format
   Appendix B: Vicid Program Source (Over)

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT: ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS

21 ABSTRACT SECURITY CLASSIFICATION: Unclassified

22a NAME OF RESPONSIBLE INDIVIDUAL:

22b TELEPHONE (Include Area Code):

22c OFFICE SYMBOL:

DD Form 1473, JUN 86 Previous editions are obsolete SECURITY CLASSIFICATION OF THIS PAGE

Abstract (continued)

Part 2

9. Appendix D: common program source
10. Appendix E: ctimer program source
11. Appendix F: declare program source
12. Appendix G: equivalence program source
13. Appendix H: etime program source
14. Appendix I: initial program source
15. Appendix J: namelist program source
16. Appendix K: network program source
17. Appendix L: structure program source
18. Appendix M: usage program source

Part 3

19. Appendix N: EXOSIM 2.0 (End-to-end)

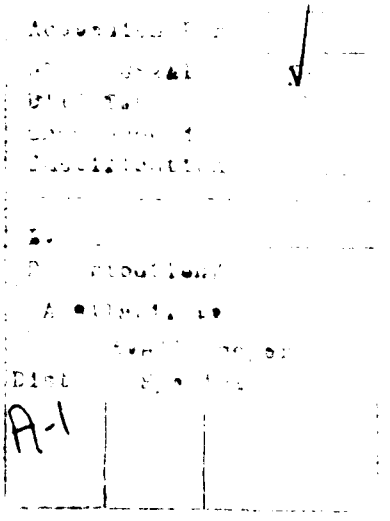
DISCLAIMER

DISCLAIMER STATEMENT - The views, opinions, and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other official documentation.

DISTRIBUTION CONTROL

(1) DISTRIBUTION STATEMENT - Approved for public release; distribution is unlimited.

(2) This material may be reproduced by or for the U.S. Government pursuant to the copyright license under the clause at DFARS 252.227 - 7013, October 1988.



ANNUAL REPORT

VOLUME 1
PART 1

TASK 1: DIGITAL EMULATION TECHNOLOGY LABORATORY

September 27, 1991

Authors

Thomas R. Collins and Stephen R. Wachtel

COMPUTER ENGINEERING RESEARCH LABORATORY

Georgia Institute of Technology

Atlanta, Georgia 30332 - 0540

Eugene L. Sanders	Cecil O. Alford
USASDC	Georgia Tech
Contract Monitor	Project Director

TABLE OF CONTENTS

PART 1

PART 2

PART 3

LIST OF FIGURES

1. Introduction

The Digital Emulation Technology Laboratory (formerly referred to as the KEW Digital Emulation Laboratory) is a principal unit within the Computer Engineering Research Laboratory (CERL) at Georgia Tech. This report addresses the objectives, requirements, and schedule of the Digital Emulation Technology Laboratory (DETL), relative to contract number DASG60-89-C-0142. This contract concerns primarily activity associated with the effort to develop an integrated hardware and software environment for end-to-end emulations of exoatmospheric interceptors such as EXOSIM. This includes the Georgia Tech Parallel Function Processor (PFP) system software for the PFP (utilities and parallel programming tools), and application software (EXOSIM). Some discussion of interfaces to specialized external hardware like the Seeker Scene Emulator (SSE) will also be included.

1.1. Objectives

Within DETL, there are two main hardware systems: the Parallel Function Processor (PFP) and the Seeker Scene Emulator (SSE). Each of these systems is a complex parallel processor, designed to function together as an emulation facility for kinetic energy weapons systems. Software development is also an active area of research, both at the system level (compilers, loaders, graphics development) and at the application level (simulation and emulation studies).

The principal objectives of DETL are as follows:

- Provide facilities for 6-DOF KEW emulation
- Provide real-time capability in excess of 2000 Hz
- Provide real-time emulation of IR FPA seekers
- Test and verify GN&C software and hardware systems
- Educate new PFP users and provide technical support.

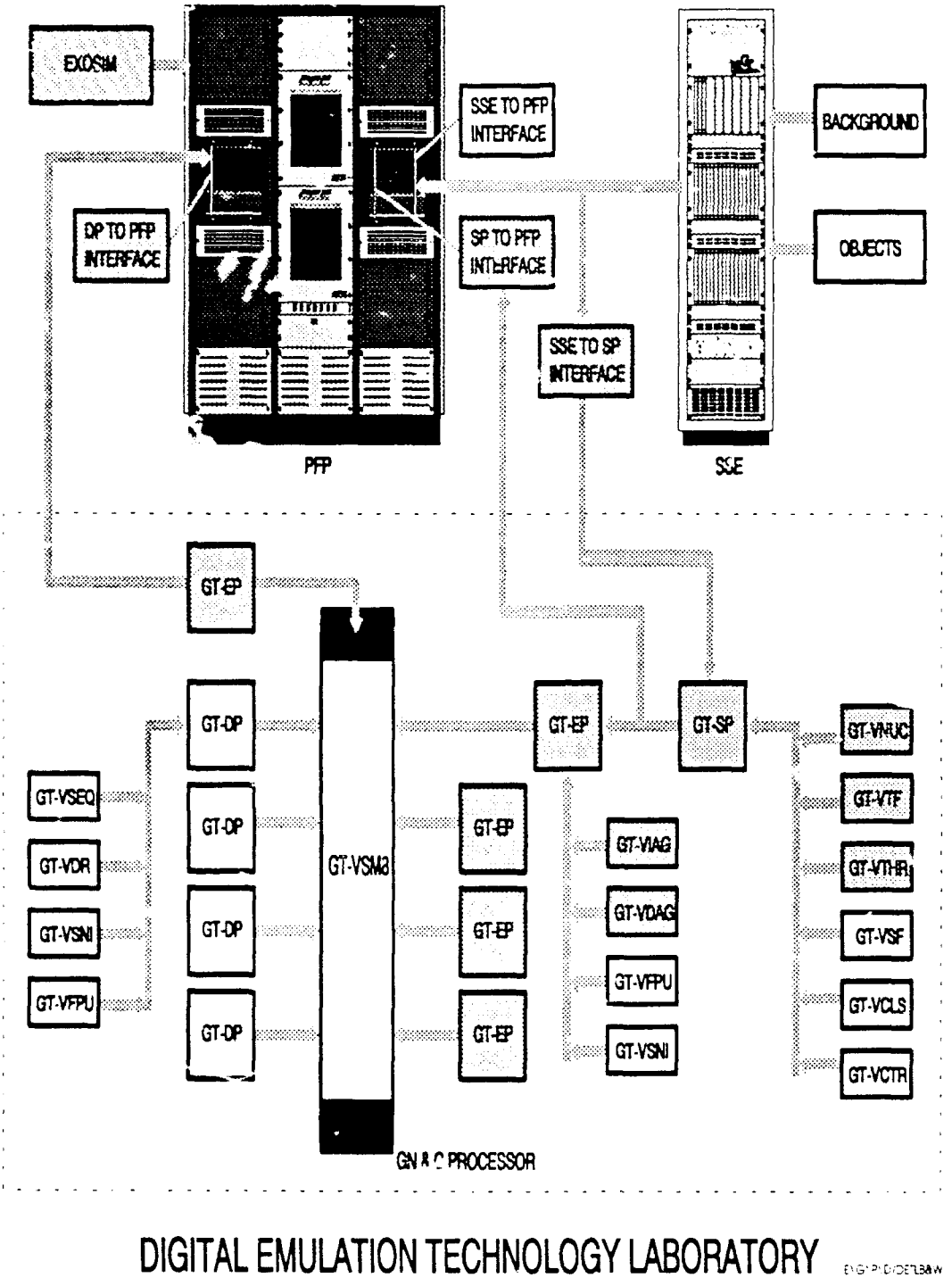


Figure 1.1: Major components of DETL

The major components used in meeting these objectives include the PFP, SSE, and associated conventional computers for basic support functions. Not all of these components are required for every task. For example, much of the ongoing research consists of running simulations (sometimes real-time, sometimes not) on the PFP, with no attached systems. This limited mode

of operation is capable of verifying missile simulation models and control laws, as well as many types of signal processing.

To provide realistic imagery in real-time, however, the Seeker Scene Emulator is required. This system generates image data as though it were coming directly off of the elements of a focal-plane array, with the scene information determined by the relative location of the simulated missile system to the targets and decoys. Additional detail on the Seeker Scene Emulator may be found in Volume 2 of this annual report.

Actual flight hardware may be tested within this system, as indicated by Figure 1.1. Most of the items contained in the lower half of this figure represent VLSI components that may be tested within DETL. The GT-DP blocks, for example, are chips for guidance and control processing that are being developed at Georgia Tech. Similarly, the GT-SP block contains signal-processing components developed at Georgia Tech. By equipping the hardware with appropriate interfaces to the PFP, the simulated functions of the GN&C Processor can migrate from the PFP to the actual hardware. These interfaces are also shown in the figure. Additional detail on the VLSI components themselves may be found in Volume 4 of this annual report.

1.1.1. GN&C Test and Evaluation -- EXOSIM

The principle objective of DETL has always been to provide a facility in which guidance, navigation, and control algorithms can be run at high speeds in order to assess their performance. Recently, this has been served by implementing EXOSIM in various forms. EXOSIM is a simulation of a representative exoatmospheric interceptor (ERIS baseline) which has evolved from several earlier simulations, including KWEST and KEERIS. Unlike KWEST, which was written in a combination of ACSL and FORTRAN, EXOSIM is written entirely in FORTRAN. Unfortunately, the programming model for EXOSIM was not especially suited for a parallel implementation, since it utilized an event-driven structure. This technique is often used to enhance the performance of discrete-event simulations on single-processor systems, since it eliminates the need to model small increments of time in which essentially nothing changes. For a continuous system, however, there is little advantage in using an event-driven structure.

One of the subcontractors for this work (Dynetics) modified Version 1.0 of EXOSIM, changing it from an event-driven structure to a time-driven structure. At the same time, it was made into an unclassified version by replacing the data set and changing two routines. This modified version of EXOSIM was first implemented at DETL and was described in the annual report for this task in FY 1990. Briefly, we generated a set of guidelines for partitioning FORTRAN code on the PFP and described a means of testing the partitions on a single-processor system. Following these guidelines, Dynetics first produced a first-stage boost version of the modified EXOSIM, partitioned for four processors. This program is called BOOST1. They then produced a first/second-stage boost version (BOOST2), partitioned for five processors. Both of these programs ran correctly on the PFP, requiring only a simple procedure of splitting up the main program along documented partitions and adding the appropriate communication instructions (which is an automated process).

BOOST2 was subsequently altered at DETL in order to extract more parallelism, thus using more processors. Since the time of the last annual report, a version has been developed which runs on 27 processors at a speed of 4 times real time (slower than real time by that factor). This version used the 80386-based processors, which are not the fastest processors available for the PFP. Then, this version was ported to the newer Sun-hosted PFP, populated with a mix of 80386 processors and the AMD 29325/7-based FPP and FPX processor boards. This allowed the simulation to run in real time.

The greatest thrust of the development effort during the past year, however, has been to analyze, debug, and partition the newer version 2.0 of EXOSIM, running end-to-end (boost, midcourse, and terminal modes of flight). This is described in detail within this report. Briefly, the basic steps were:

1. Convert the event-driven structure to a time-driven structure more suitable for the PFP,
2. Debug this single-processor version to produce a portable version, removing VAX dependencies and uninitialized variables in the process,
3. Partition the code in stages, improving execution time, using 80386-based processors,
4. Minimize double-precision requirements, and
5. Port some partitions to the FPP and FPX boards to achieve real-time operation.

At this time, we are occupied with step 5, writing new compilers and tools to more fully utilize the available processors. In the interest of demonstrating real-time performance of EXOSIM 2.0, a boost-phase-only version of the partitioned program was spun off as a side effort and is now running in real time. This complements an earlier midcourse/terminal-phase-only version which was demonstrated in July 1991, running real-time in conjunction with the SSE and described in volume 3 of this annual report. Taken together, these two versions (boost-phase-only and midcourse/terminal-only) do not constitute an end-to-end simulation, since the midcourse/terminal version only runs with preset data values.

1.1.2. Education and Technical Support

The Digital Emulation Technology Laboratory first presented a class on the programming and operation of the PFP in December 1989. During the past year, a PFP has been delivered to the KDEC facility at USASDC in Huntsville, Alabama. To support this facility, another two-day PFP class was presented at KDEC, using their PFP, in April 1991. As before, the students were employees of USASDC and its contractors. The class included material on parallel processing fundamentals, the PFP model of parallelism, PFP hardware, the host operating system, and typical applications. Approximately three-fourths of the time was used for hands-on experience with the PFP, a 50% increase from our first class, based on the opinions of our earlier participants.

To address the needs and concerns of potential PFP users at KDEC, DETL provided a technical briefing on the PFP on July 26, 1991. This briefing was given to a blue-ribbon committee

reporting to Dr. E. L. Wilkinson through Doyce Satterfield, covered the PFP hardware, system software, basic operation, software utilities, and application areas.

We also organized a technical committee, the Parallel Simulation Technology Working Group. This group includes members from SDC-affiliated companies who can meet to discuss simulation techniques, general parallel programming topics, PFP issues, and ongoing SDC simulation work. The first meeting took place on August 15, 1990. The presentation topics at that meeting are listed below.

Unique PFP Programming Considerations

Automatic crossbar/sequencer code generation (S. Wachtel -- Georgia Tech)

EXOSEEK Seeker Simulation (R. Stone -- BDM)

Parallel Simulation Techniques

Carriers, Threads, and Event Multi-Tasking Capabilities (W. Tan - Georgia Tech)

Extraction of lower-level parallelism in EXOSIM (C. O. Alford/P. Bingham -- Georgia Tech)

Parallel Simulation Applications

Vehicle simulation requirements for scene generation (K. Smith -- Sentar)

Implementation status of EXOSIM on the PFP (T. Collins -- Georgia Tech)

Signal-processing Algorithms (H. Gatzke -- TBE)

1.2. Schedules and milestones

As of August 1991, there are four 32-processor PFP systems available. Two of these are available for classified operation. One of these two secure machines, hosted by the Intel RMX-based host, is populated with mostly 80386-based processors, but also has one FPP available and several 286-based processors to fill up the slots. The other secure system is populated with up to six FPX processors, up to four 80386-based processors, and up to 23 FPP processors. The other two systems are the 286-based machine located at KDEC and the FPP-based machine for internal development of FPP/Sun host software. Not included is a prototype Multibus II PFP.

The unsecured PFPs (at DETL and KDEC) both include the basic packaging and power supplies to support expansion to 64-processor capability. The 386-based PFP may eventually be paired with the Multibus II PFP to produce a 64-processor hybrid system.

The major milestones completed over the period of this report are as follows:

- Integration of additional 386/12 processors into PFP, making some available for the mostly-FPP/FPX PFP (to handle memory-intensive applications),

- Development of utility software on the RMX host,

- Development of new system software, such as loaders,

- Upgrades to the Floating-Point Processor (FPP) Compiler,

- Development of parallel-processing support utilities, including one that analyzes program structure, one that assists in automated timing charts, and one which checks and generates communication code,

- Enhancement of libraries of communication procedures for processor-processor and processor-host interaction, providing uniform interfaces across several languages (C, Fortran, Pascal, and PL/M),

- Improved layout of a new "piggyback" board to provide crossbar communcation capability to the 386/12 boards through their iSBX interfaces,

- Presentation of offsite education in PFP programming,

- Extensive development of various versions of EXOSIM 1.0 and 2.0, and

- Demonstration of the 256-processor Seeker Scene Emulator generating frames in real time and in closed-loop with EXOSIM 2.0.

The most significant causes for delays during the past year have been

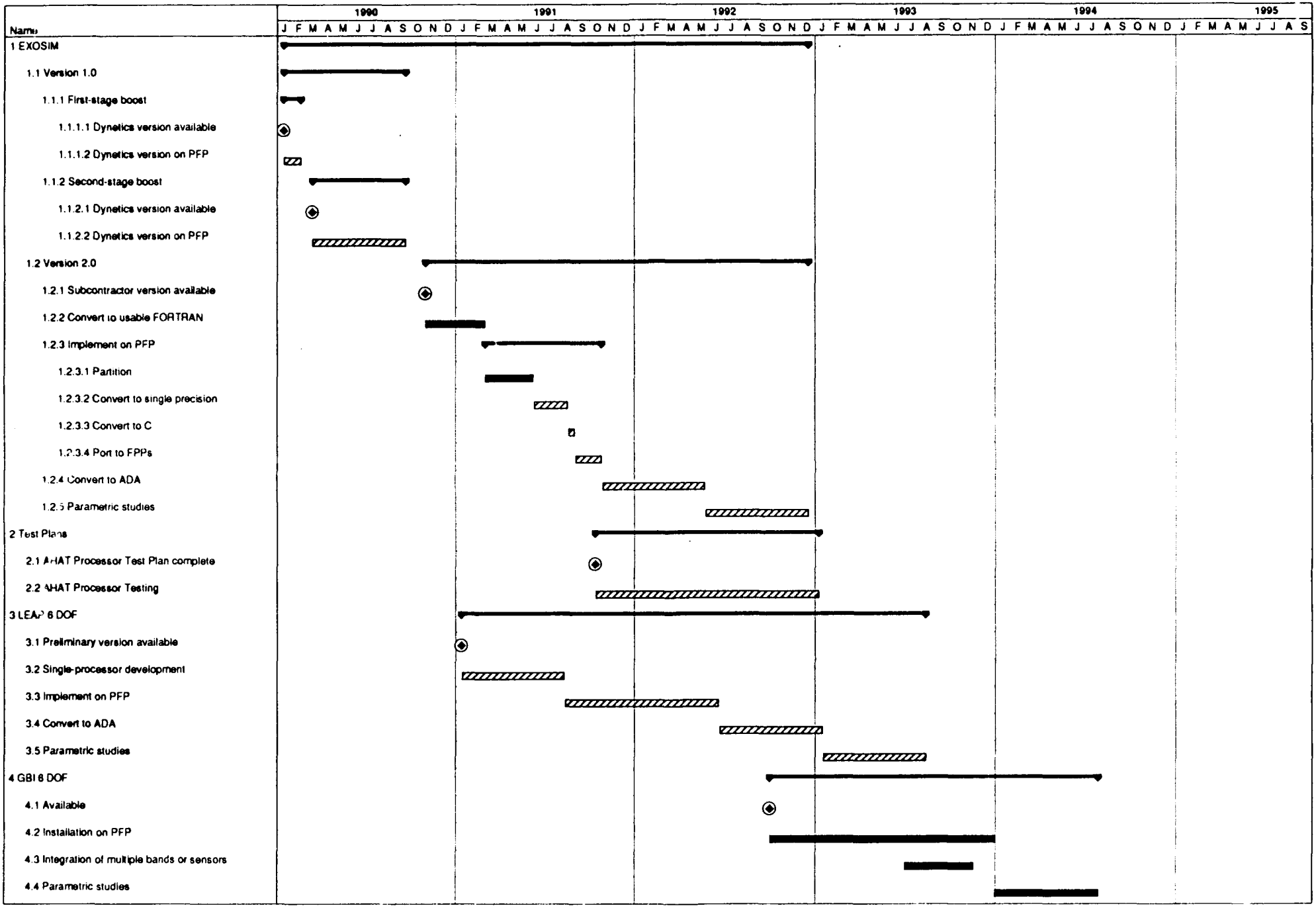
- Inadequate compilers for the FPP and FPX boards,

- Insufficient memory on the FPP and FPX boards, and

- Lack of programming support personnel.

During the coming year, the highest priority will be placed on the optimal use of DETL technologies in simulation applications such as EXOSIM. The schedule for this effort is shown in Figure 1.2. The implementation dates for midcourse and end-to-end EXOSIM are generally consistent with earlier estimates. Dates for LEAP and GBI implementation are estimates and assume that these simulations are available to be ported to the PFP.

Figure 1.2: DETL Schedule



The remainder of this report will describe the hardware and software associated with the Digital Emulation Technology Laboratory, with an emphasis on the work completed during the previous contract year. The hardware information includes updated status of the PFP units, new processors, host enhancements, and communication interfaces. A brief description is also given for the physical facility itself and some auxiliary computers contained within. The software information includes new versions of utilities which support the GT-FPP (Floating-Point Processor) and the GT-XSD and GT-SEQ (crossbar and sequencer boards), as well as updates to application software (EXOSIM and LEAP).

Based on comments and suggestions from outside users and members of the blue-ribbon panel mentioned above, DETL feels a need to address requirements for dynamic, on-demand communication between processors, a feature not supported on the current PFP. While this limitation can be worked around in the typical PFP applications of the past, including EXOSIM and several foreseeable interceptor simulations, it will become more serious in applications such as battle-management. Although Georgia Tech's concepts for an advanced PFP (the APFP), address these limitations, it is also desirable to enhance existing systems, including the PFP at KDEC. Consequently, we have a conceptual design for a new crossbar which will eliminate the need for a sequencer, along with all of its limitations.

Although we have already developed a complete set of software tools that meet our own research requirements, there is a need for general programming aids, particularly for users not accustomed to specialized computers. This also became clear based upon interaction with KDEC users and the blue-ribbon panel. Several of the tools which DETL has used internally will be developed further and released to outside users.

2. Hardware and Facilities

This section begins with a description of the Parallel Function Processor, including recent changes, and then discusses the current configuration of the two alternative host computers. Most of the detail, though, is devoted to recent improvements and current board status.

2.1. Parallel Function Processor (PFP)

The Special-Purpose Operational Computing Kernel, or SPOCK, evolved from a Ph.D. dissertation (by James O. Hamblen) on a new architecture designed to solve ballistic missile simulations. Before digital computers came into prominence, some of these simulations had been performed quite effectively on analog computers, in which basic circuit elements are interconnected by a patch panel to create an approximation to the real system.

Digital computers provided the potential of much higher accuracy in the simulations, but at the cost of speed: most real systems could not be simulated nearly as fast as they really run, generally referred to as *real time*. In 1978, Georgia Tech's SPOCK I addressed the problem by showing how up to 6 processors could effectively perform such a simulation.

Building on the previous experience, in 1982 a prototype of a 32-processor system, SPOCK II, demonstrated greater capability with more-powerful processors. In addition to the digital processors, SPOCK II also had analog input and output channels. This provided the important capability of interfacing seamlessly with the external environment, for real-time control of analog systems.

Since that time we have developed SPOCK II into the Parallel Function Processor (PFP), a fully-operational testbed for simulation and emulation problems from both military and nonmilitary applications. The architecture never stagnates -- the original Intel 8086/8087 processors were each roughly as powerful as an IBM PC, but now they can be replaced with any of four newer processors. One is based on the Intel 80286/80287 and performs as well as an IBM AT. Another is based on the Intel 80386/80387, and the last two are based on the AMD 29325 and 29327 processors and are about 25-100 times faster for the floating-point calculations which it is designed to perform. Integration of 80486-based processors is planned for the coming year, which should provide near-FPP/FPX speed, with significantly more memory and extensive software support.

All of the processors, or Parallel Processing Elements (PPEs), support the 16-by-16 crossbar interconnection, allowing each to communicate directly with the others. Multiple conversations may take place simultaneously on the crossbar, and it is also possible for a single processor to broadcast data to every other processor in a single instruction cycle. Since the crossbar has been reduced in size from a full 19-inch rack down to a cluster of eight circuit boards, it is now possible to have the power of 32 minicomputers in two racks, and still have all of the processors work together efficiently.

Each of the current processors has two interfaces: one to the crossbar for data communication while running, and one to a shared bus that is used for loading programs and data from a central host. Virtually any imaginable processor can be fitted to a processor slot in the PFP. In fact, if an image-processing problem was part of a larger simulation problem, it could be assigned to an array processor within the PFP system. Co-processing boards have been developed at Georgia Tech that evaluate complex floating-point functions in a fraction of the time used by the best supercomputers on the market today. These co-processing boards "piggy-back" on the processors described earlier.

Similarly, a complete minicomputer system with an attached 3-D graphics workstation has been connected to one of the PFP processors, thus effectively becoming a part of the multiprocessor system. This allows sophisticated graphics to be generated in real time as the simulations proceed.

These enhancements demonstrate that other architectures can be applied *as needed* within the enveloping PFP architecture. But there is also a way to increase the PFP's capability at a higher level. Since the number of processing nodes in a crossbar is practically limited because of the large number of switches required, the PFP needs a way to grow beyond its crossbar. A fully-operational interconnection board has been developed which occupies a processor node in a single PFP system. When a processor communicates with this interconnection board, the data is passed out over an external channel to an identical board in another complete PFP. By adding more interconnections, multiple PFPs may form a higher level of parallel processing. A triangle of three PFPs still allows each processor to communicate with any other processor with no intervening processors, although there may be some waiting for an available channel.

The standard configuration of the PFP at this time is a 64-processor system (2 crossbars), packaged in a three-rack system, including the host. A single-crossbar system can be packaged in two smaller racks, if desired. Both the Intel RMX-based 310 host and the Sun Unix-based 386i host are currently supported.

2.1.1. Physical Description

The full 64-node PFP, complete with the host computer, occupies three 19 inch wide by 32 inch deep by 75 inch high equipment racks. Each outer rack contains 32 PPE slots. The center rack contains the two crossbars, two sequencers, the host computer and two crossbar status displays.

All processors, as well as the sequencer, conform to Intel's Multibus I specification. They are connected to the host through a custom Multibus repeater system, which is used by the host to communicate with each PPE. Each 16 by 16 crossbar switch is made from four 8 by 8 switch boards connected through a custom backplane. Each 8 by 8 switch board is built to a 15.75 inch by 14.44 inch Eurocard standard. Both crossbars are housed in one 19 inch wide card cage.

Each of the 64 nodes in the system is occupied by a PPE. A PPE can be one of five different boards; an array interconnect, an Intel 80286-based commercially available processor, an Intel 80386-based commercially available processor, a Georgia Tech Floating-Point Processor, or a

multi-channel analog I/O interface. Other boards will be developed as necessary to enhance the capability of the PFP.

The Georgia Tech Floating Point Processor (GT-FPP/3) is an 8 MFLOP computing engine based on the AMD 29325 floating point chip. Currently, the board is programmed using a subset of Pascal or in C. Ada and FORTRAN are supported by Ada-to-C and FORTRAN-to-C converter programs. An enhanced version, the GT-FPX board, is based on the AMD 29327 and supports double precision arithmetic along with a wider range of integer and control operations.

The iSBC286/12 processor is commercially built by the Intel corporation. It is a cheaper, lower performance board than the GT-FPP/3. The board is useful in applications that require large amounts of memory such as table look ups. Presently, most of the programming is done in FORTRAN and C , although Pascal, PL/M, and other Intel standard utilities are available. The crossbar interface to this board is built to fit the Intel standard iSBX port. Supporting other Multibus I processors that have this port only requires changes in the board's firmware. The iSBC386/12 processor is an 80386-based equivalent of the iSBC286/12 board, with approximately a 3-4 times speed improvement for typical PFP applications. Several iSBC 486/12 boards are on order. These are 80486-based replacements for the 80286 and 80386-based boards, and should be roughly as fast as the FPP/FPX processor boards, while supporting more onboard memory and more compilers.

The analog input/output board consists of four analog to digital input channels and four digital to analog output channels. The output portion consists of 4 separate digital to analog converters. The input portion consists of 4 sample and hold circuits multiplexed through one analog to digital converter. Any combination of inputs and outputs are available for use. All digital conversions have 12 significant bits.

The array interconnect board (GT-ARI/1) is used as a direct interconnect between crossbars. Each array interconnect may send and receive 16 bit words simultaneously from other array interconnects. The use of array interconnects affects only the crossbar code and is otherwise transparent to the individual processors.

All programs are written and compiled on the host computer then downloaded to the processors. Currently, each problem is analyzed by a programmer and split into parts which are then compiled for individual processors. A separate compiler is used to load the crossbar and sequencer with the instructions for processor communication.

The major components of a full system are:

1. The host machine. (This may be an Intel 310 or Sun 386i)

2. An MDB Systems Data Shuttle 2000 removable disk drive unit.

3. Up to 64 processors and array interconnects, in any combination.

4. Up to two sequencers.

5. Up to two full 16 by 16 GT-XB/2 crossbar switches.

6. Up to two GT-XSD/2 status display units.

7. Up to two equipment racks containing Multibus I card cages, sequencer cabling, and power distribution.

8. One equipment rack containing the crossbar, sequencers, crossbar status displays, and appropriate power distribution.

2.1.2. Intel 310 Host

The Intel 310 host is based on a 12 Mhz 80286 processor (actually the same 286/12 board available for use in the PFP) and runs the Intel iRMX operating system. We have also replaced the host 286/12 board with a 386/12 board, in much the same way that we have replaced the PFP 286/12 processors with 386/12 processors. This configuration can execute computationally-intensive applications (including compilation and linking) about four times faster that the 286-based host. The host is tied to the PFP through a custom set of repeater boards developed here at Georgia Tech. A master repeater board is located within the host chassis, and slave repeater boards are located within the racks of processors. The machine supports all standard Intel languages running under the iRMX operating system, including C, Pascal, PLM, and FORTRAN. Programs written in any of these languages may be compiled and linked on the host and then downloaded to processor boards (iSBC 286/12s or iSBC 386/12s) in the PFP for execution. In addition, the host supports a compiler that implements a subset of Pascal for use with the GT-FPP/3 custom floating-point processor.

2.1.3. Sun 386i Host

The Sun 386i host is based on a 25 Mhz 80386 processor and runs the Unix operating system. It is the basis of an eventual replacement for the Intel 310, leading to higher performance and a more user-friendly environment. The hardware interface to the PFP is similar to that of the Intel 310 host, except that the master repeater board is located within a dedicated Multibus rack, connected to the Sun host by a PC-to-Multibus link. (The Sun 386i utilizes the PC/AT bus.) A C compiler has been written to support the GT-FPP/3 processor, and other languages will be supported via translators (Ada-to-C and FORTRAN-to-C). All low-level drivers interfacing the Sun to the PFP are complete and several Fortran, Ada, and C programs have been loaded and tested, including versions of EXOSIM. The Sun also supports standard Intel-supplied languages for programming the iSBC 386/12 processors.

2.2. Seeker Scene Emulator (SSE)

In addition to developing crossbar machines like the PFP, DETL is actively studying other architectures, since there is no such thing as a completely general-purpose parallel computer. One of the most promising is a group of architectures built around a new microprocessor chip, the Inmos Transputer. Unlike previous microprocessors, the Transputer was specifically designed to be interconnected with others of its kind. Since a single chip includes the processor,

memory, and communication ports, it is possible to build a parallel machine with little more than a group of Transputers.

Each Transputer has four links that can be used to tie them together, allowing a wide range of architectures to be built. One of our principal applications for the Transputer is a Seeker Scene Emulator, a machine that models what an imaging sensor on a missile would see during a mission. Most simulations of such systems tend to simplify the infrared sensing process in order to minimize computations, but the Georgia Tech Seeker Scene Emulator will provide a signal which can be displayed on a screen and will look virtually identical to a real view of an incoming threat.

This seeker output can then be used by a simulation running on the PFP, or by an actual guidance and control processor, like the one being developed for our VLSI devices. The Seeker Scene Emulator will use 256 Transputers, so when connected to PFP in a simulation, it will be another example of a specialized parallel processor within the more general crossbar architecture of PFP.

Under direction from the U. S. Army Strategic Defense Command, the Computer Engineering and Research Laboratory at the Georgia Institute of Technology and BDM Corporation are developing a real-time Focal Plane Array Seeker Scene Emulator. This unit will enhance Georgia Tech's capabilities in KEW system testing and performance demonstration.

The FPA Seeker Scene Emulator combines advanced hardware developed at Georgia Tech with a BDM-generated database to produce signals based upon target radiometric information, seeker optical characterization, FPA detector characterization, and simulated background environments. Using real-time, positional updates, typically from the Georgia Tech Parallel Function Processor, the Seeker Scene Emulator can combine elements of the pre-computed database to form an image that is positionally and radiometrically correct.

In conjunction with development of the FPA Seeker Scene Emulator, research into signal processing of seeker data is underway. The Seeker Scene Emulator provides a platform for the expedient testing of algorithms and implementations. Currently, a parallel-processing network is being used to test various signal processing "building blocks."

Detailed information about the Seeker Scene Emulator may be found in Volume 2 of this annual report.

2.3. Other computer systems

Originally, a Digital Equipment Corporation MicroVAX II was used as the primary file server for the Seeker Scene Emulator, but this function has now been transferred to the Sun 386i which also serves as one of the PFP host machines. The MicroVAX can still be used to transfer programs and data to and from other contractors. Programs written for VAXes and other off-site computers may be loaded onto this MicroVAX via its nine-track tape drive. From there, files may be transferred to the PFP hosts (Intel 310s or Sun 386i's) or to other computer systems. Also, additional simulation support is available on this system through the MatrixX and ACSL languages. Both languages provide an environment for the simulation of discrete and

continuous-time systems, including a choice of integration methods. MatrixX also has a graphical user interface for entering simulation specifics. This MicroVAX is approved for classified data processing. This system is equipped with a nine-track tape system, the standard TK50 tape unit, an Ethernet network interface, and a Caplin Cybernetics Corporation QT0 Transputer Interface Module.

Another MicroVAX is dedicated to a Chromatics 3-D graphics workstation. This combination of machines may be directly connected to a PFP processor in order to display complex three-dimensional graphics during simulations. Both of these machines are approved for classified data processing. In order to improve graphics quality and to support standard computing platforms, a Silicon Graphics Indigo workstation is currently in the purchasing plans. This machine would replace the Chromatics system for high-quality graphics output, while also supporting the graphics requirements of the SSE.

A secure Ultrix machine was required to run the PFP programming tools that have been developed under Ultrix. Consequently, the Chromatics MicroVAX can now be brought up as a secure Ultrix machine. Ultrix V4.0 has been installed onto removable disk canisters ready for use whenever we can get secure code and data to the machine. This machine has also been useful to shake out the portability problems of the code, using both the UNIX FORTRAN compiler and the VMS FORTRAN compiler.

It was also necessary to build a secure PC disk for the an IBM-compatible PC, using a 20MB Bernoulli disk. The system has been used to transfer secure data via the network from the iRMXII host or VAX VMS host (via OpenNET) to the PC or the Ultrix machine (via TCP/IP).

2.4. Secure laboratory

The principal elements of the Digital Emulation Technology Laboratory are housed in a laboratory on the third floor of the Centennial Research Building which has been approved for classified operation up to the secret level. Within this facility are most of the machines which have been described, including:

- the 80386/80286-based PFP (32-processor), with FPP capability,
- the FPP/FPX/80386-based PFP (32-processor),
- two RMX-based PFP host machines (Intel 310s),
- one Unix-based PFP host machine (Sun 386i),
- the Seeker Scene Emulator,
- the MicroVAX with 9-track and TK50 tape drives,
- the MicroVAX/Chromatics system, and
- an IBM-compatitble PC serving as the SSE host.

Each of these machines is approved for classified processing. The two PFP host machines are functionally identical, with one always available as a backup. A safe is also provided for storage of classified documents and magnetic media. All classified hard disks are removable, and the classified operating disks are stored in the safe.

3. FPP/FPX Development Tools

3.1. Introduction

This section covers the latest changes and additions which have been made to the FPP/FPX processor development tools.

3.2. FPP/FPX object module loader

As part of our effect to improve the software support for the FPP/FPX processors, the old FPP/FPX object module loader has been replaced by a new object module loader. Refer to Appendix B for the complete program source.

The old object module loader took as input a list of relocatable FPP or FPX object modules and constructed a corresponding absolute FPP or FPX load module. Each object module was assumed to be required and so was relocated to an absolute address according to the order in which they appeared on the input list. It was limited to 50 object modules maximum.

```
FPP example:

        vicld \
        /vol/pfp/lib/fppheld.fppo \
        ../library_1/subroutine_1.fppo \
        ../library_1/     ...
        ../library_1/subroutine_n.fppo \
             ...
        ../library_n/subroutine_1.fppo \
        ../library_n/     ...
        ../library_n/subroutine_n.fppo \
        /vol/pfp/lib/fpptail.fppo

FPX example:
        vicld \
        /vol/pfp/lib/fpphead.fpxo \
        ../library_1/subroutine_1.fpxo \
        ../library_1/     ...
        ../library_1/subroutine_n.fpxo \
             ...
        ../library_n/subroutine_1.fpxo \
        ../library_n/     ...
        ../library_n/subroutine_n.fpxo \
        /vol/pfp/lib/fpptail.fpxo
```

Figure 3.1: Example use of old object module loader.

The new object module loader take as input a list of relocatable FPP or FPX object modules and constructs a corresponding absolute FPP or FPX load module. But, the new object module loader will only assume that the first object module is required an that the remaining object modules should only be included to satisfy a code or data dependency requirement. The required object modules will then be relocated to an absolute address according to the order in which they appear on the input list. It is currently limited to 1024 object modules maximum.

```
FPP example:
        vicld \
        /vol/pfp/lib/fpphead.fppo \
        ../library_1/*.fppo \
            ...
        ../library_n/*.fppo \
        /vol/pfp/lib/fpptail.fppo

FPX example:
        vicld \
        /vol/pfp/lib/fpphead.fpxo \
        ../library_1/*.fpxo \
            ...
        ../library_n/*.fpxo \
        /vol/pfp/lib/fpptail.fpxo
```

Figure 3.2: Example use of new object module loader.

The new object module loader is also faster than the old object module loader because it reads each object module once where the old loader read each object module three times.

3.3. FPP/FPX program downloader

The PFP FPP/FPX downloader program takes the output of the FPP/FPX object module loader program and downloads the code and data into a target FPP or FPX processor. Refer to Appendix C for the complete program source.

The command line syntax is:

loadfpp <processor$_1$>=<file$_1$> [... <processor$_n$>=<file$_n$>]

where:

<processor$_i$> = target processor name

<file$_i$> = host file name

The FPP/FPX program downloader performs the following steps:

1. Build a bootstrap program for downloading the application program data.
2. Download the bootstrap program.
3. Start the bootstrap program.
4. Send the application program data to the bootstrap program.
5. Stop the bootstrap program.
6. Download the application program code.
7. Start the application program.

4. Software Development Tools

4.1. Introduction

The following software development tools consist of a collection of programs developed at the Georgia Institute of Technology. These tools, which execute under either SUN OS or Ultrix, were made to assist the PFP user in the design, development and analysis of programs for the PFP.

4.2. Sequential programming tools

This section will discuss programs designed to assist the PFP user in the design, development and analysis of sequential programs.

4.2.1. INITIAL program

The purpose of the INITIAL program is to determine if any uninitialized variables exist in a FORTRAN 77 program. Refer to Appendix I for the complete program source.

The command line syntax is:

initial <input file> <output file> [-conditional=y or n]

where:

<input file> = input file name
<output file> = output file name

The INITIAL program with the option "-conditional=y" determines whether a variable is uninitialized by assuming the following about the program control flow:

1. that execution proceeds sequentially through a subprogram from top to bottom.

2. when a subprogram call is encountered, control is passed to that subprogram with the resulting changes in the formal arguments reflected back through the callers actual arguments.

3. that data and parameter statement assignments always occur.

4. that variable references and assignments outside conditionals always occur.

5. that variable assignments inside conditionals always occur.

6. that variable references inside conditionals always occur.

The INITIAL program with the option "-conditional=n" determines whether a variable is uninitialized by assuming the following about the program control flow:

1. that execution proceeds sequentially through the program from top to bottom.

2. when a subprogram call is encountered, control is passed to that subprogram with the resulting changes in the formal arguments reflected back through the callers actual arguments.

3. that data and parameter statement assignments always occur.

4. that variable references and assignments outside conditionals always occur.

5. that variable assignments inside conditionals never occur.

6. that variable references inside conditionals always occur.

From these two choices, the option "-conditional=n" implements the most conservative approach in determining whether a variable is uninitialized.

R- or {R}	Reference without set
S- or {S}	Set without reference
CR- or {CR}	Conditional Reference without set
CS- or {CS}	Conditional Set without reference
RS or {RS}	Reference and then Set
SR or {SR}	Set and then Reference
CRS or {CRS}	Conditional Reference and then Set
CSR or {CSR}	Conditional Set and then Reference

Figure 4.1: INITIAL codes.

The following figures will be used to demonstrate the INITIAL program.

```
default:        example.1 example.2


example.1:      example.f
        initial example.f example.1 -conditional=y


example.2:      example.f
        initial example.f example.2 -conditional=n
```

Figure 4.2: INITIAL example makefile.

```
        PROGRAM example
        DATA a /100./
        CALL sub1(a, b, c, d, e)
        f = a + b + c + d + e + g
        CALL square(f)
        END

        SUBROUTINE sub1(a, b, c, d, e)
        IF (a .LE. 10.) THEN
          b = 0.
        ELSE
          b = b + 1
        END IF
        c = 0.
        CALL sub2(a, d, e)
        END

        SUBROUTINE sub2(a, d, e)
        IF (a .LE. 5.) THEN
          CALL square(d)
        END IF
        CALL sub3(a, e)
        END

        SUBROUTINE sub3(a, e)
        IF (a .LE. 1) e = 0.
        END

        SUBROUTINE square(z)
        z = z**2
        END
```

Figure 4.3: INITIAL example input (EXAMPLE.F).

The following figure contains output produced by the INITIAL program for example 1. The variables "D" and "G" in subprogram "EXAMPLE" have been spotted by INITIAL as potential uninitialized variables. The variable "D" has the code "RS" which means that it's being referenced before it's being set. The variable "G" has the code "R-" which means that it's being referenced without ever being set. The other lines in the output show each successive subprogram with its formal arguments and local variables, if any.

```
EXAMPLE
D RS
G R-

SUB1 A(R)=1,B(SR)=2,C(S)=3,D(RS)=4,E(S)=5

SUB2 A(R)=1,D(RS)=2,E(S)=3

SUB3 A(R)=1,E(S)=2

SQUARE Z(RS)=1
```

Figure 4.4: INITIAL example 1. output (EXAMPLE.1).

The following figure contains output produced by the INITIAL program for example 2. The variables "B", "D" and "G" in subprogram "EXAMPLE" have been spotted by INITIAL as potential uninitialized variables. The variables "B" and "D" have the code "CRS" which means that they are being conditionally referenced before being set. The variable "G" has the code "R-" which means that it's being referenced without ever being set. The other lines in the output show each successive subprogram with its formal arguments and local variables, if any.

```
EXAMPLE
B CRS
D CRS
G R-

SUB1 A(R)=1,B(CRS)=2,C(S)=3,D(CRS)=4,E(S)=5

SUB2 A(R)=1,D(CRS)=2,E(S)=3

SUB3 A(R)=1,E(S)=2

SQUARE Z(RS)=1
```

Figure 4.5: INITIAL example 2. output (EXAMPLE.2).

Limitations of the INITIAL program:

1. Equivalenced variables are not supported.

2. Common block variables are not supported.

3. If goto statements are used in such a way to violate the above assumptions, then the results that the INITIAL program produces may not be correct.

4.2.2. DECLARE program

The DECLARE program takes as input a FORTRAN 77 program and produces as output a complete set of FORTRAN 77 declaration and data statements. Refer to Appendix F for the complete program source.

The command line syntax is:

declare <input file> <output file> [-initialize=n or y]

where:

<input file> = input file name
<output file> = output file name

The DECLARE program with the option "-initialize=n" will parse the input FORTRAN 77 and produce the following output for each subprogram:

1. a subprogram skeleton.
2. then the formal argument declaration statements, if any.
3. then the common block declaration statements, if any.
4. then the variable declaration statements, if any.

The DECLARE program with the option "-initialize=y" will parse the input FORTRAN 77 and produce the following output for each subprogram:

1. a subprogram skeleton.
2. then the formal argument declaration statements, if any.
3. the common block declaration statements, if any.
4. then the variable declaration statements, if any.
5. then the data statements for initialized variables, if any.
6. then the data statements for uninitialized variables, if any.

Also, scalar variables or arrays that are declared in a subprogram but not used referenced or set will be excluded from the output FORTRAN 77 declaration and data statements.

The following figures will be used to demonstrate the DECLARE program.

```
default:        example.1 example.2


example.1:      example.f
        declare example.f example.1 -initialize=n


example.2:      example.f
        declare example.f example.2 -initialize=y
```

Figure 4.6: DECLARE example makefile.

```
        PROGRAM example
        IMPLICIT INTEGER(a-z)
        DIMENSION a(10), b(10), c(10)

        CALL sub1(c)
        CALL sub2(c)
        END

        SUBROUTINE sub1(c)
        IMPLICIT INTEGER(a-z)
        DIMENSION c(10)
        REAL i, j(10), k
        COMMON /block/ i(10), j, k(10)
        DATA aa /1/

        DO 10 a = aa, 10
          c(a) = c(a) + i(a) + j(a) + k(a)
10      CONTINUE
        END

        SUBROUTINE sub2(c)
        IMPLICIT INTEGER(a-z)
        DIMENSION c(10)
        REAL i(10), j, k(10)
        COMMON /block/ i, j(10), k
        DATA bb /1/

        DO 10 b = bb, 10
          i(b) = k(b) - c(b)
          j(b) = j(b) - c(b)
          k(b) = i(b) - c(b)
10      CONTINUE
        END
```

Figure 4.7: DECLARE example input (EXAMPLE.F).

From the following output, you can see that DECLARE program recognized that the variables "A" (in "EXAMPLE") and "B" (in "EXAMPLE") were not necessary and so were excluded from the output. Also, note that implicit variable declarations were changed to explicit variable declarations.

```
      PROGRAM EXAMPLE
* VARIABLE DECLARATION
      INTEGER*4 C(10)
      END

      SUBROUTINE SUB1()
* FORMAL ARGUMENT DECLARATION
      INTEGER*4 C(10)
* COMMON /BLOCK/ DECLARATION
      COMMON /BLOCK/ I,J,K
      REAL*4 I(10)
      REAL*4 J(10)
      REAL*4 K(10)
* VARIABLE DECLARATION
      INTEGER*4 AA
      INTEGER*4 A
      END

      SUBROUTINE SUB2()
* FORMAL ARGUMENT DECLARATION
      INTEGER*4 C(10)
* COMMON /BLOCK/ DECLARATION
      COMMON /BLOCK/ I,J,K
      REAL*4 I(10)
      REAL*4 J(10)
      REAL*4 K(10)
* VARIABLE DECLARATION
      INTEGER*4 BB
      INTEGER*4 B
      END
```

Figure 4.8: DECLARE example 1. output (EXAMPLE.1).

From the following output, you can see that the DECLARE program recognized that the variables "A" (in "EXAMPLE"), "B" (in "SUB1") and "C" (in "SUB2") did not have an initial value. Therefore, each variable was given an initial value of zero. Also, the subprogram "BLKDAT" was automatically included to initialize all elements within the common "BLOCK".

```
      PROGRAM EXAMPLE
* VARIABLE DECLARATION
      INTEGER*4 C(10)
* UNINITIALIZED DATA
      DATA C /10 * 0/
      END

      SUBROUTINE SUB1()
* FORMAL ARGUMENT DECLARATION
      INTEGER*4 C(10)
* COMMON /BLOCK/ DECLARATION
      COMMON /BLOCK/ I,J,K
      REAL*4 I(10)
      REAL*4 J(10)
      REAL*4 K(10)
* VARIABLE DECLARATION
      INTEGER*4 AA
      INTEGER*4 A
* INITIALIZED DATA
      DATA AA /1/
* UNINITIALIZED DATA
      DATA A /0/
      END

      SUBROUTINE SUB2()
* FORMAL ARGUMENT DECLARATION
      INTEGER*4 C(10)
* COMMON /BLOCK/ DECLARATION
      COMMON /BLOCK/ I,J,K
      REAL*4 I(10)
      REAL*4 J(10)
      REAL*4 K(10)
* VARIABLE DECLARATION
      INTEGER*4 BB
      INTEGER*4 B
* INITIALIZED DATA
      DATA BB /1/
* UNINITIALIZED DATA
      DATA B /0/
      END

      BLOCK DATA BLKDAT
* COMMON /BLOCK/ DECLARATION
      COMMON /BLOCK/ I,J,K
      REAL*4 I(10)
      REAL*4 J(10)
      REAL*4 K(10)
* COMMON /BLOCK/ INITIALIZATION
      DATA I /10 * 0E0/
      DATA J /10 * 0E0/
      DATA K /10 * 0E0/
      END
```

Figure 4.9: DECLARE example 2. output (EXAMPLE.2).

Limitations of the DECLARE program:

1. Equivalenced variables are not supported.

2. Parameters are not supported in array declarations.

4.2.3. STRUCTURE program

The STRUCTURE program analyzes a FORTRAN 77 program in order to generate a FORTRAN 77 subprogram call structure. Refer to Appendix L for the complete program source.

The command line syntax is:

structure <input file> <output file>

where:

<input file> = input file name
<output file> = output file name

The following figures will be used to demonstrate the STRUCTURE program.

```
default:        example.out

example.out:    example.f
        structure example.f example.out
```

Figure 4.10: STRUCTURE example makefile.

```
        PROGRAM example
        DATA dt /1./
        t = 0.
        x = 0.
        y = 0.
        z = 0.

10      CONTINUE
        CALL send(x, y, z)
        altitude = sqrt(x**2 + y**2 + z**2)
        CALL receive(dx, dy, dz)
        CALL integrate(x, dx, dt)
        CALL integrate(y, dy, dt)
        CALL integrate(z, dz, dt)
        t = t + dt
        GO TO 10
        END

        SUBROUTINE send(x, y, z)
        CALL send_real_32bit(x)
        CALL send_real_32bit(y)
        CALL send_real_32bit(z)
        END

        SUBROUTINE receive(dx, dy, dz)
        CALL receive_real_32bit(dx)
        CALL receive_real_32bit(dy)
        CALL receive_real_32bit(dz)
        END

        SUBROUTINE integrate(x, dx, dt)
        x = x + dx*dt
        END
```

Figure 4.11: STRUCTURE example input (EXAMPLE.F).

The following figure contains output produced by the STRUCTURE program. As you can see, the STRUCTURE program determined that the subprogram "example" called subprograms "send", "sqrt", "receive" and "integrate". When the subprogram "send" was called, it called

subprogram "send_real_32bit" three times. When the subprogram "receive" was called, it called subprogram "receive_real_32bit" three times.

```
example
        send
                send_real_32bit
                send_real_32bit
                send_real_32bit
        sqrt
        receive
                receive_real_32bit
                receive_real_32bit
                receive_real_32bit
        integrate

        integrate
```

Figure 4.12: STRUCTURE example output (EXAMPLE.OUT).

There are no limitations on the STRUCTURE program.

4.2.4. CTIMER program

The CTIMER program analyzes a FORTRAN 77 program in order to produce a serial program time profile from timing subprogram calls. Refer to Appendix E for the complete program source.

The command line syntax is:

ctimer <input file> <output file>

where:

<input file> = input file name
<output file> = output file name

The CTIMER program takes as input a FORTRAN 77 program and produces as output a modified FORTRAN 77 program with timer code automatically inserted around subprogram calls. This modified program is then compiled, bound, and executed on the PFP to produce an output file which details the number of times and length of time spent in each subprogram call.

The following figures will be used to demonstrate the CTIMER program.

```
default:        example.f ctimer.txt


example.f:      example.f.old
        ctimer example.f.old example.f >ctimer.txt
```

Figure 4.13: CTIMER example makefile.

```
        PROGRAM example
        DATA dt /1./
        t = 0.
        x = 0.
        y = 0.
        z = 0.

*LOOP* PROLOGUE

10      CONTINUE
*LOOP* START

        CALL send(x, y, z)
        altitude = sqrt(x**2 + y**2 + z**2)
        CALL receive(dx, dy, dz)
        CALL integrate(x, dx, dt)
        CALL integrate(y, dy, dt)
        CALL integrate(z, dz, dt)
        t = t + dt

*LOOP* STOP
        IF (t .LE. 1000.) GO TO 10

*LOOP* EPILOGUE
        END

        SUBROUTINE send(x, y, z)
        CALL send_real_32bit(x)
        CALL send_real_32bit(y)
        CALL send_real_32bit(z)
        END

        SUBROUTINE receive(dx, dy, dz)
        CALL receive_real_32bit(dx)
        CALL receive_real_32bit(dy)
        CALL receive_real_32bit(dz)
        END

        SUBROUTINE integrate(x, dx, dt)
        x = x + dx*dt
        END
```

Figure 4.14: CTIMER example input (EXAMPLE.F.OLD).

The following figure contains the FORTRAN 77 program created by the CTIMER program. The required "*LOOP* PROLOGUE" and "*LOOP* EPILOGUE" comments have been replaced by calls to timer routines and the optional "*LOOP* START" and "*LOOP* STOP" comments have been removed. Note that the "CALL start_timer()" and "CALL stop_timer()" now appear around each subprogram call.

```
      PROGRAM example
      DATA dt /1./
      t = 0.
      x = 0.
      y = 0.
      z = 0.

      CALL timer_prologue()

10    CONTINUE

      CALL start_timer(1)
      CALL send(x, y, z)
      CALL stop_timer(1)
      altitude = sqrt(x**2 + y**. + z**2)
      CALL start_timer(2)
      CALL receive(dx, dy, dz)
      CALL stop_timer(2)
      CALL start_timer(3)
      CALL integrate(x, dx, dt)
      CALL stop_timer(3)
      CALL start_timer(4)
      CALL integrate(y, dy, dt)
      CALL stop_timer(4)
      CALL start_timer(5)
      CALL integrate(z, dz, dt)
      CALL stop_timer(5)
      t = t + dt

      IF (t .LE. 1000.) THEN
        GO TO 10
      END IF

      CALL timer_epilogue()
      END

      SUBROUTINE send(x, y, z)
      CALL start_timer(6)
      CALL send_real_32bit(x)
      CALL stop_timer(6)
      CALL start_timer(7)
      CALL send_real_32bit(y)
      CALL stop_timer(7)
      CALL start_timer(8)
      CALL send_real_32bit(z)
      CALL stop_timer(8)
      END

      SUBROUTINE receive(dx, dy, dz)
      CALL start_timer(9)
      CALL receive_real_32bit(dx)
      CALL stop_timer(9)
      CALL start_timer(10)
      CALL receive_real_32bit(dy)
      CALL stop_timer(10)
      CALL start_timer(11)
      CALL receive_real_32bit(dz)
      CALL stop_timer(11)
      END

      SUBROUTINE integrate(x, dx, dt)
      x = x + dx*dt
      END
```

Figure 4.15: CTIMER example output (EXAMPLE.F).

The following figure contains the timer list also created by the CTIMER program. The list contains three fields:

1. the timer number.

2. the calling subprogram name.

3. The called subprogram name.

```
1 example send
2 example receive
3 example integrate
4 example integrate
5 example integrate
6 send send_real_32bit
7 send send_real_32bit
8 send send_real_32bit
9 receive receive_real_32bit
10 receive receive_real_32bit
11 receive receive_real_32bit
```

Figure 4.16: CTIMER example output (CTIMER.TXT).

The following figure contains the output from executing the EXAMPLE.F program on the PFP. The output contains four fields:

1. the timer number.

2. the comment "TIMER".

3. the number of times the subprogram was called.

4. the length of time spent inside the subprogram call.

```
1 TIMER <count1> <time1>
2 TIMER <count2> <time2>
3 TIMER <count3> <time3>
4 TIMER <count4> <time4>
5 TIMER <count5> <time5>
6 TIMER <count6> <time6>
7 TIMER <count7> <time7>
8 TIMER <count8> <time8>
9 TIMER <count9> <time9>
10 TIMER <count10> <time10>
11 TIMER <count11> <time11>
```

Figure 4.17: CTIMER example output (CTIMER.OUT).

Limitations of the CTIMER program:

1. Function subprograms are not timed.

2. Function statements are not timed.

4.3. Parallel programming tools

This section will discuss programs designed to assist the PFP user in the design, development and analysis of parallel programs for the PFP.

4.3.1. NETWORK program

The NETWORK program analyzes multiple FORTRAN 77 programs in order to automatically generate a crossbar/sequencer compiler program. Refer to Appendix K for the complete program source.

The NETWORK program takes as input multiple FORTRAN 77 programs and produces a crossbar/sequencer compiler program with the maximum number of overlapping transfers per cycle as possible. The number of cycles and number of overlaps are dependant on the order of the manually placed SENDs and RECEIVEs in the input FORTRAN 77 programs and the group identity and ordering priority from the PRIORITY.TXT file. The NETWORK program orders variables by looking for all variables that are ready to be sent and picking the highest ordering priority variables first. Also, it uses the group identity to make sure that only variables within a group are allowed to overlap. The group identity and ordering priority information are easily obtained from the Microsoft Project timing analysis charts.

An additional task of the NETWORK program is to verify the integrity of the network communication:

1. The sending processor's variable name and type must match exactly with the receiving processor's variable name and type.

2. All processor's receive FIFOs are examined to make sure that variable order matches the sending variable order.

3. All SENDs and/or RECEIVEs must match, i. e., no leftovers.

The following figures will be used to demonstrate the NETWORK program.

```
default:        network.1 network.2


COMMUNICATION =  \
        block0.communication \
        block1.communication \
        block2.communication \
        block3.communication


PROCESSOR =  \
        00=block0.communication \
        01=block1.communication \
        02=block2.communication \
        03=block3.communication


network.1: priority.1 $(COMMUNICATION)
        network <priority.1 $(PROCESSOR) >network.1


network.2: priority.2 $(COMMUNICATION)
        network <priority.2 $(PROCESSOR) >network.2


.SUFFIXES: .f .communication
.f.communication:
        communication $*.f $*.communication
```

Figure 4.18: NETWORK example makefile.

```
        PROGRAM block0
        DATA a /1./
        DATA t /0./

*LOOP* PROLOGUE

10      CONTINUE
*LOOP* START

        CALL send_real_32bit(a)
        CALL receive_real_32bit(c)
        CALL receive_real_32bit(d)
        a = c + d
        t = t + 1.

*LOOP* STOP
        IF (t .LT. 1000.) GO TO 10

*LOOP* EPILOGUE
        END
```

Figure 4.19: NETWORK example input (BLOCK0.F).

```
        PROGRAM block1
        DATA b /1./
        DATA t /0./

*LOOP* PROLOGUE

10      CONTINUE
*LOOP* START

        CALL receive_real_32bit(a)
        CALL send_real_32bit(b)
        CALL receive_real_32bit(d)
        b = a + d
        t = t + 1.

*LOOP* STOP
        IF (t .LT. 1000.) GO TO 10

*LOOP* EPILOGUE
        END
```

Figure 4.20: NETWORK example input (BLOCK1.F).

```
        PROGRAM block2
        DATA c /1./
        DATA t /0./

*LOOP* PROLOGUE

10      CONTINUE
*LOOP* START

        CALL receive_real_32bit(a)
        CALL send_real_32bit(c)
        CALL receive_real_32bit(d)
        c = a + d
        t = t + 1.

*LOOP* STOP
        IF (t .LT. 1000.) GO TO 10

*LOOP* EPILOGUE
        END
```

Figure 4.21: NETWORK example input (BLOCK2.F).

```
        PROGRAM block3
        DATA d /1./
        DATA t /0./

*LOOP* PROLOGUE

10      CONTINUE
*LOOP* START

        CALL receive_real_32bit(a)
        CALL receive_real_32bit(b)
        CALL send_real_32bit(d)
        d = a + b
        t = t + 1.

*LOOP* STOP
        IF (t .LT. 1000.) GO TO 10

*LOOP* EPILOGUE
        END
```

Figure 4.22: NETWORK example input (BLOCK3.F).

The following figure contains the group identity and ordering priority for example 1. The NETWORK program will try to, if possible, order the transfers with variable "A" first, then "C", then "B" and finally "D". Since there are not any group dividers, there is only one group.

```
A
C
B
D
```

Figure 4.23: NETWORK example 1. input (PRIORITY.1).

The following figure contains the crossbar/sequencer compiler program for example 1. Note that the NETWORK program determined that the "C" and "B" transfers could be overlapped. Also, the bottom four lines are included to summarize the communication requirements for each program.

```
LOOP

CYCLE [ 1 ]
  p01, p02, p03  :=  p00.2; [ REAL*4 A 1000 ]

CYCLE [ 2 ]
  p00  :=  p02.2; [ REAL*4 C 1001 ]
  p03  :=  p01.2; [ REAL*4 B 1002 ]

CYCLE [ 3 ]
  p00, p01, p02  :=  p03.2; [ REAL*4 D 1003 ]

[ p00 = block0.f, S =   1, R =   2,   3 ]
[ p01 = block1.f, S =   1, R =   2,   3 ]
[ p02 = block2.f, S =   1, R =   2,   3 ]
[ p03 = block3.f, S =   1, R =   2,   3 ]
```

Figure 4.24: NETWORK example 1. output (NETWORK.1).

The following figure contains the group identity and ordering priority for example 2. The NETWORK program will try to, if possible, order the transfers, with variable "A" first, then "C", then "B" and finally "D". But this time, there are two groups divided by the "#" character. Group 1 contains the variables "A" first and then "C". Group 2 contains variables "B" first and "D".

```
A
C
#
B
D
```

Figure 4.25: NETWORK example 2. input (PRIORITY.2).

The following figure contains the crossbar/sequencer compiler program for example 2. Note that the NETWORK program determined that the "C" and "B" transfers could not be overlapped since they each belong to different groups. Also, the bottom four lines are included to summarize the communication requirements for each program.

```
LOOP

CYCLE [ 1 ]
  p01, p02, p03  :=  p00.2; [ REAL*4 A 1000 ]

CYCLE [ 2 ]
  p00  :=  p02.2; [ REAL*4 C 1001 ]

CYCLE [ 3 ]
  p03  :=  p01.2; [ REAL*4 B 2000 ]

CYCLE [ 4 ]
  p00, p01, p02  :=  p03.2; [ REAL*4 D 2001 ]

[ p00 = block0.f, S =   1, R =   2,   3 ]
[ p01 = block1.f, S =   1, R =   2,   3 ]
[ p02 = block2.f, S =   1, R =   2,   3 ]
[ p03 = block3.f, S =   1, R =   2,   3 ]
```

Figure 4.26: NETWORK example 2. output (NETWORK.2).

Limitation(s):

1. A maximum of 1000 groups with a maximum of 1000 variables per group.

2. Variables which are communicated between programs must maintain the same name and type.

```
* INCORRECT
        PROGRAM BLOCK0
        CALL SEND_REAL_32BIT( A )
        CALL SEND_REAL_64BIT( B )
        END

        PROGRAM BLOCK1
        CALL RECEIVE_REAL_64BIT( A )
        CALL RECEIVE_REAL_64BIT( BB )
        END

* CORRECT
        PROGRAM BLOCK0
        CALL SEND_REAL_32BIT( A )
        CALL SEND_REAL_64BIT( B )
        END

        PROGRAM BLOCK1
        CALL RECEIVE_REAL_32BIT( A )
        CALL RECEIVE_REAL_64BIT( B )
        END
```

Figure 4.27: NETWORK limitation 2. example.

3. A SEND and/or RECEIVE cannot be duplicated within a program.

```
* INCORRECT
        IF ( A. LT. B ) THEN
                C = ...
                CALL SEND_REAL_32BIT( C )
        ELSE
                C = ...
                CALL SEND_REAL_32BIT( C )
        END IF

* CORRECT
        IF ( A. LT. B ) THEN
                C = ...
        ELSE
                C = ...
        END IF
        CALL SEND_REAL_32BIT( C )
```

Figure 4.28: NETWORK limitation 3. example.

4. A SEND and/or RECEIVE must be used every cycle.

```
* INCORRECT
        IF ( A. LT. B ) THEN
                C = ...
                CALL SEND_REAL_32BIT( C )
        END IF

* CORRECT
        IF ( A. LT. B ) THEN
                C = ...
        END IF
        CALL SEND_REAL_32BIT( C )
```

Figure 4.29: NETWORK limitation 4. example.

5. There is no guarantee the crossbar/sequencer compiler code is "optimal" since the problem is NP-complete but every effect has been made to make it as efficient as possible.

4.3.2. USAGE program

The USAGE program analyzes multiple FORTRAN 77 programs in order to automatically identify variables requiring interprocessor communication. Refer to Appendix M for the complete program source.

The USAGE program takes as input multiple FORTRAN 77 programs and produces an output which summarize all variables that share the same name and type and are used in more than one program. Interprocessor variables which are only referenced in a program but never set are output with the code "R". Interprocessor variables which are set or set and referenced in a program are output with the code "S".

The USAGE program will make the following rules in determining variable usage:

1. variable usage will only be checked between the required "*LOOP* PROLOGUE" and "*LOOP* EILOGUE" comments.

2. when a subprogram call is encountered, control is passed to that subprogram with the resulting changes in the formal arguments reflected back to in the callers actual arguments.

3. that data and parameter statement assignments always occur.

4. that variable assignments inside and outside conditionals always occur.

5. that variable references inside and outside conditionals always occur.

The following figures will be used to demonstrate the USAGE program.

```
default:        summary.txt


USAGE = \
        block0.usage \
        block1.usage \
        block2.usage \
        block3.usage


PROCESSOR = \
        00=block0.usage \
        01=block1.usage \
        02=block2.usage \
        03=block3.usage


summary.txt:    combine.txt
        summary $(PROCESSOR) <combine.txt >summary.txt


combine.txt:    type.txt $(USAGE)
        combine $(PROCESSOR) <type.txt >combine.txt


type.txt:       example.f
        type example.f type.txt


.SUFFIXES: .f .usage
.f.usage:
        usage $*.f $*.usage
```

Figure 4.30: USAGE example makefile.

```
        PROGRAM block0

        REAL x, y, z, t
        REAL dx, dy, dz, dt
        REAL altitude

        DATA x, y, z, t /0., 0., 0., 0./
        DATA dx, dy, dz, dt /1., 1., 1., 1./

*LOOP* PROLOGUE

10      CONTINUE
*LOOP* START

        CALL send_real_32bit(x)
        x = x + dt*dx
        CALL receive_real_32bit(dz)
        dx = dx + (0.1*dz)
        t = t + dt

*LOOP* STOP
        IF (t .LT. 1000.) GO TO 10

*LOOP* EPILOGUE
        END
```

Figure 4.31: USAGE example input (BLOCK0.F).

```
        PROGRAM block1

        REAL x, y, z, t
        REAL dx, dy, dz, dt
        REAL altitude

        DATA x, y, z, t /0., 0., 0., 0./
        DATA dx, dy, dz, dt /1., 1., 1., 1./

*LOOP* PROLOGUE

10      CONTINUE
*LOOP* START

        CALL send_real_32bit(y)
        y = y + dt*dy
        CALL receive_real_32bit(dz)
        dy = dy + (0.1*dz)
        t = t + dt

*LOOP* STOP
        IF (t .LT. 1000.) GO TO 10

*LOOP* EPILOGUE
        END
```

Figure 4.32: USAGE example input (BLOCK1.F).

```
        PROGRAM block2

        REAL x, y, z, t
        REAL dx, dy, dz, dt
        REAL altitude

        DATA x, y, z, t /0., 0., 0., 0./
        DATA dx, dy, dz, dt /1., 1., 1., 1./

*LOOP* PROLOGUE

10      CONTINUE
*LOOP* START

        CALL send_real_32bit(z)
        CALL send_real_32bit(dz)
        z = z + dt*dz
        dz = dz + (0.1*dz)
        t = t + dt

*LOOP* STOP
        IF (t .LT. 1000.) GO TO 10

*LOOP* EPILOGUE
        END
```

Figure 4.33: USAGE example input (BLOCK2.F).

```
        PROGRAM block3

        REAL x, y, z, t
        REAL dx, dy, dz, dt
        REAL altitude

        DATA x, y, z, t /0., 0., 0., 0./
        DATA dx, dy, dz, dt /1., 1., 1., 1./

*LOOP* PROLOGUE

10      CONTINUE
*LOOP* START

        CALL receive_real_32bit(x)
        CALL receive_real_32bit(y)
        CALL receive_real_32bit(z)
        altitude = sqrt(x**2 + y**2 + z**2)
        t = t + dt

*LOOP* STOP
        IF (t .LT. 1000.) GO TO 10

*LOOP* EPILOGUE
        END
```

Figure 4.34: USAGE example input (BLOCK3.F).

The following figure contains the output from the USAGE program produced after analyzing the four input programs. A total of 5 variables were determined by USAGE to require interprocessor communication. Note that the "T" line ends with "WARNING". It is there to inform the user that the USAGE program recognized multiple assignments of the same variable, a potential problem. In this case, the variable is a "replicated" variable. This means that the code that changes this variable is identical each place it is set and consequently the warning may be ignored. Finally, the bottom four lines are included to summarize the communication requirements for each program.

```
Page  1                          |P00|P01|P02|P03|P04|P05|P06|P07|P08|P09|...|P31|
---------------------------------+---+---+---+---+---+---+---+---+---+---+---+---+
DZ                REAL*4         | R | R | S |   |   |   |   |   |   |   |...|   |
---------------------------------+---+---+---+---+---+---+---+---+---+---+---+---+
T                 REAL*4         | S | S | S | S |   |   |   |   |   |   |...|   |WARNING
---------------------------------+---+---+---+---+---+---+---+---+---+---+---+---+
X                 REAL*4         | S |   |   | R |   |   |   |   |   |   |...|   |
---------------------------------+---+---+---+---+---+---+---+---+---+---+---+---+
Y                 REAL*4         |   | S |   | R |   |   |   |   |   |   |...|   |
---------------------------------+---+---+---+---+---+---+---+---+---+---+---+---+
Z                 REAL*4         |   |   | S | R |   |   |   |   |   |   |...|   |
---------------------------------+---+---+---+---+---+---+---+---+---+---+---+---+

p00 = block0.f, S =   2, R =   1,   3
p01 = block1.f, S =   2, R =   1,   3
p02 = block2.f, S =   3, R =   0,   3
p03 = block3.f, S =   1, R =   3,   4
```

Figure 4.35: USAGE example output (SUMMARY.TXT).

Limitations of the USAGE program:

1. Equivalenced variables are not supported.

2. Common block variables are not supported.

4.3.3. ETIMER program

The ETIMER program analyzes multiple FORTRAN 77 programs in order to produce a parallel program time profile from timing computation and I/O events. Refer to Appendix H for the complete program source.

The ETIMER program takes as input multiple FORTRAN 77 programs and produces as output modified FORTRAN 77 programs with timer code automatically inserted around each computation block that is bounded by a SEND and/or RECEIVE or by the required "*LOOP* START" and "*LOOP* STOP" comments. The ETIMER program also produces an output file which contains unique event numbers for the project, each program, each computation block and SEND and RECEIVE pair. This output, when combined with the actual times from the 286/386 real-time timers and formatted properly, can be input into Microsoft Project which will produce an accurate representation of a parallel simulation.

There are two ways of determining the dependency information required by Microsoft Project:

1. Assume that the SEND FIFO is not full. Consequently, the SEND processors execution is not-blocked but the RECEIVE processors execution is blocked until the SEND occurs. The problem with the timing analysis charts produced by Microsoft Project with this dependency information is that it is probably optimistic with respect to the amount of parallelism shown.

2. Assume that the SEND FIFO is full. Consequently, the SEND and RECEIVE processors execution is blocked until the RECEIVE occurs. The problem with the timing analysis charts produced by Microsoft Project with this dependency information is that it is probably pessimistic with respect to the amount of parallelism shown.

Unfortunately, because of the finite-length of FIFOs, at some times assumption 1 is valid, and some times assumption 2 is valid. One way to model finite-length FIFO activity correctly is to time the SENDs. This way, if a FIFO were full, the time will be large, otherwise, the time will be small.

After experimentation, we decided to do timing analysis on EXOSIM 2.0 using assumption 1 with the number of 16-bit crossbar transfers as the time taken to do a SEND .

The following figures will be used to demonstrate the ETIMER program.

```
default:        $(NEW) event.txt


OLD = \
        block0.f.old \
        block1.f.old \
        block2.f.old \
        block3.f.old


NEW = \
        block0.f \
        block1.f \
        block2.f \
        block3.f


$(NEW) event.txt: $(OLD)
        cat $(OLD) | etimer | fsplit
        mv main000.f event.txt
```

Figure 4.36: ETIMER example makefile.

The following four output files were output from the ETIMER program.

```
        PROGRAM block0

        REAL x, y, z, t
        REAL dx, dy, dz, dt
        REAL altitude

        DATA x, y, z, t /0., 0., 0., 0./
        DATA dx, dy, dz, dt /1., 1., 1., 1./

        CALL timer_prologue()

10      CONTINUE

        CALL send_real_32bit(x)
        CALL start_timer(4)
        x = x + dt*dx
        CALL stop_timer(4)
        CALL receive_real_32bit(dz)
        CALL start_timer(5)
        dx = dx + (0.1*dz)
        t = t + dt

        CALL stop_timer(5)
        IF (t .LT. 1000.) GO TO 10

        CALL timer_epilogue()
        END
```

Figure 4.37: ETIMER example output (BLOCK0.F).

```
      PROGRAM block1

      REAL x, y, z, t
      REAL dx, dy, dz, dt
      REAL altitude

      DATA x, y, z, t /0., 0., 0., 0./
      DATA dx, dy, dz, dt /1., 1., 1., 1./

      CALL timer_prologue()

10    CONTINUE

      CALL send_real_32bit(y)
      CALL start_timer(8)
      y = y + dt*dy
      CALL stop_timer(8)
      CALL receive_real_32bit(dz)
      CALL start_timer(9)
      dy = dy + (0.1*dz)
      t = t + dt

      CALL stop_timer(9)
      IF (t .LT. 1000.) GO TO 10

      CALL timer_epilogue()
      END
```

Figure 4.38: ETIMER example output (BLOCK1.F).

```
      PROGRAM block2

      REAL x, y, z, t
      REAL dx, dy, dz, dt
      REAL altitude

      DATA x, y, z, t /0., 0., 0., 0./
      DATA dx, dy, dz, dt /1., 1., 1., 1./

      CALL timer_prologue()

10    CONTINUE

      CALL send_real_32bit(z)
      CALL send_real_32bit(dz)
      CALL start_timer(13)
      z = z + dt*dz
      dz = dz + (0.1*dz)
      t = t + dt

      CALL stop_timer(13)
      IF (t .LT. 1000.) GO TO 10

      CALL timer_epilogue()
      END
```

Figure 4.39: ETIMER example output (BLOCK2.F).

```
      PROGRAM block3

      REAL x, y, z, t
      REAL dx, dy, dz, dt
      REAL altitude

      DATA x, y, z, t /0., 0., 0., 0./
      DATA dx, dy, dz, dt /1., 1., 1., 1./

      CALL timer_prologue()

10    CONTINUE

      CALL receive_real_32bit(x)
      CALL receive_real_32bit(y)
      CALL receive_real_32bit(z)
      CALL start_timer(15)
      altitude = sqrt(x**2 + y**2 + z**2)
      t = t + dt

      CALL stop_timer(15)
      IF (t .LT. 1000.) GO TO 10

      CALL timer_epilogue()
      END
```

Figure 4.40. ETIMER example output (BLOCK3.F).

The following figure contains the output from the ETIMER program produced after analyzing the four input programs.

```
1 project project 1 ""
2 program block0 2 ""
3 communication x 1 real*4 3 ""
4 computation ? 3 "3"
5 computation ? 3 "4,12"
6 program block1 2 ""
7 communication y 1 real*4 3 ""
8 computation ? 3 "7"
9 computation ? 3 "8,12"
10 program block2 2 ""
11 communication z 1 real*4 3 ""
12 communication dz 1 real*4 3 "11"
13 computation ? 3 "12"
14 program block3 2 ""
15 computation ? 3 "3,7,11"
```

Figure 4.41: ETIMER example output (ETIMER.TXT).

The next figure contains the output after executing the four BLOCK.F programs on the PFP. The output contains the following fields:

1. the timer number.
2. the comment "TIMER".
3. the length of time spent in a computation event during simulation time windowa.
4. the length of time spent in a computation event during simulation time windowb.
5. the length of time spent in a computation event during simulation time windowc.

This information, when combined with ETIMER.TXT, and formatted properly, is input into Microsoft Project which then will produce an accurate representation of a parallel simulation in the form of a Gannt chart for each simulation time window.

```
4 TIMER <time4a> <time4b> <time4c> ...
5 TIMER <time5a> <time5b> <time5c> ...
8 TIMER <time8a> <time8b> <time8c> ...
9 TIMER <time9a> <time9a> <time9c> ...
13 TIMER <time13a> <time13b> <time13c> ...
15 TIMER <time15a> <time15b> <time15c> ...
```

Figure 4.42: ETIMER example output (ETIMER.OUT).

Limitations of the ETIMER program:

1. The ETIMER program requires an output program (like Microsoft Project) in order to display and manipulate the Gannt chart.

4.4. Special purpose tools

This section will discuss programs designed to assist us in the transformation of the sequential program EXOSIM 2.0 into a form suitable for porting to the PFP.

4.4.1. NAMELIST program

The purpose of the NAMELIST program is to transform the EXOSIM 2.0 namelist files into FORTRAN 77 data statements. Refer to Appendix J for the complete program source.

This program was necessary since namelist statements are not valid FORTRAN 77. Also, we wanted to eliminate as much host I/O as possible in order to make the program as machine independent as possible.

The command line syntax is:

namelist <input file> <output file>

where:

<input file> = input file name
<output file> = output file name

The following figures will be used to demonstrate the NAMELIST program.

```
default:        example.out

example.out:    example.txt
        namelist example.txt example.out
```

Figure 4.43: NAMELIST example makefile.

```
 $CONST1
      CHAR1  = 'CONSTANT PARAMETER NAMELIST 1'

      PI = 3.14,

      DTEPS = 1.0E-6,

      IPLOT = 1,

      IPRINT = 1,

      TABLE =  1.10,  2.20,   4.400,   8.800,  16.160,
              32.32, 64.64, 128.128, 256.256, 512.512,

      ARRAY = 9*0.0, 100.0, 9*0.0, 1000.0
 $END
```

Figure 4.44: NAMELIST example input (EXAMPLE.TXT).

Using the above input, the NAMELIST program produced the following output. Note that each variable or array assignment is made into one data statement.

```
*CONST1
      DATA char1 /'CONSTANT PARAMETER NAMELIST 1'/
      DATA pi /3.14/
      DATA dteps /1.0E-6/
      DATA iplot /1/
      DATA iprint /1/
      DATA table /1.1, 2.2, 4.4, 8.8, 16.16, 32.32, 64.64, 128.128,
     & 256.256, 512.512/
      DATA array /9*0., 100., 9*0., 1000./
*END
```

Figure 4.45: NAMELIST example output (EXAMPLE.OUT).

Limitations of the NAMELIST program:

1. The NAMELIST program doesn't check variable type or array sizes, consequently, the data statements may require some manual modifications.

4.4.2. EQUIVALENCE program

The EQUIVALENCE program takes as input a FORTRAN 77 program in order to extract variable initialization information from equivalence and data statements. The output produced is

compatible with the PROLOG program explained later. Refer to Appendix G for the complete program source.

The command line syntax is:

equivalence <input file> <output file>

where:

<input file> = input file name
<output file> = output file name

The following figures will be used to demonstrate the EQUIVALENCE program.

```
default:        example.out

example.out:    example.f
        equivalence example.f example.out
```

Figure 4.46: EQUIVALENCE example makefile.

```
      SUBROUTINE SEEKER

      EQUIVALENCE ( VAR(   1) , SAMACQ     ) , ( VAR(   2) , SAMTRK     )
      EQUIVALENCE ( VAR(   3) , SAMTRM     ) , ( VAR(   4) , FOV        )
      EQUIVALENCE ( VAR(   5) , SEKNOS(1)  ) , ( VAR(  29) , SEKTIM(1)  )
      EQUIVALENCE ( VAR(  53) , QNTZP      ) , ( VAR(  54) , RATE(1)    )
      EQUIVALENCE ( VAR(  60) , SNRMIN     ) , ( VAR(  61) , FOVLIM     )
      EQUIVALENCE ( VAR(  62) , SNRACQ     ) , ( VAR(  63) , RFINAL     )
      EQUIVALENCE ( VAR(  64) ,ACQRNG(1,1)) , ( VAR(  80) , TRGSIG(1)  )
      EQUIVALENCE ( VAR(  84) , RNGTRK     ) , ( VAR(  85) , RNGTRM     )

      EQUIVALENCE (IVAR(   1) , SEKTYP     ) , (IVAR(   2) , ITRGSG     )
      EQUIVALENCE (IVAR(   3) , BCKGRD     )

      DATA NR,NI /   16,   3 /

      DATA IREAL / 2221,2222,2223,2224,2227,2251,2275,2276,2282,2283,
     1             2284,2285,2286,2302,2372,2374, 4*0 /

      DATA LREAL /    1,   1,   1,   1,  24,  24,   1,   6,   1,   1,
     1                1,   1,  16,   4,   1,   1, 4*0 /

      DATA IINT  /   41,  42,  43, 7*0 /

      DATA LINT  /    1,   1,   1, 7*0 /

      END

      SUBROUTINE TARGET

      EQUIVALENCE ( VAR(   1) , TARLEN   ) , ( VAR(   2) , TARWID   )
      EQUIVALENCE ( VAR(   3) , GMU      ) , ( VAR(   4) , TARPOS   )
      EQUIVALENCE ( VAR(   7) , TARVEL   ) , ( VAR(  10) , TARRI    )
      EQUIVALENCE ( VAR(  11) , CSOPOS   ) , ( VAR(  14) , CSOVEL   )
      EQUIVALENCE ( VAR(  17) , CSORI    ) , ( VAR(  18) , TNKPOS   )
      EQUIVALENCE ( VAR(  21) , TNKVEL   ) , ( VAR(  24) , TNKRI    )
      EQUIVALENCE ( VAR(  25) , RHOPOS   ) , ( VAR(  28) , RHOVEL   )
      EQUIVALENCE ( VAR(  31) , RHORI    ) , ( VAR(  32) , CLTPOS   )
      EQUIVALENCE ( VAR(  35) , CLTVEL   ) , ( VAR(  38) , CLTRI    )
      EQUIVALENCE ( VAR(  39) , DTR      ) , ( VAR(  40) , WIDTH    )
      EQUIVALENCE ( VAR(  41) , FOCLEN   ) , ( VAR(  42) , RMULT    )

      EQUIVALENCE (IVAR(   1) , NOBJ     ) , (IVAR(   2) , ISKOUT   )
      EQUIVALENCE (IVAR(   3) , SEKTYP   ) , (IVAR(   4) , NTARRS   )

      DATA NR,NI / 22, 4 /

      DATA IREAL / 3660,3661,   6,3662,3665,3668,3669,3672,3675,
     1             3676,3679,3682,3683,3686,3689,3690,3693,3696,
     2                2,3616,3614,3697,28*0 /

      DATA LREAL /    1,   1,   1,   3,   3,   1,   3,   3,   1,
     1                3,   3,   1,   3,   3,   1,   3,   3,   1,
     2                1,   1,   1,   1,28*0 /

      DATA IINT  /  260, 241,  41, 261, 6*0 /

      DATA LINT  /    1,   1,   1,   1, 6*0 /

      END
```

Figure 4.47: EQUIVALENCE example input (EXAMPLE.F).

Using the above figure as input (a FORTRAN 77 program fragment from EXOSIM 2.0), the EQUIVALENCE program produced the following output. The information contained about each variables includes:

1. source file name

2. local variable name

3. global variable name

4. variable usage count

```
d("UUSEEKER.FOR","SAMACQ","RIN(2221)",0)
d("UUSEEKER.FOR","SAMTRK","RIN(2222)",0)
d("UUSEEKER.FOR","SAMTRM","RIN(2223)",0)
d("UUSEEKER.FOR","FOV","RIN(2224)",0)
d("UUSEEKER.FOR","SEKNOS","RIN(2227)",0)
d("UUSEEKER.FOR","SEKTIM","RIN(2251)",0)
d("UUSEEKER.FOR","QNTZP","RIN(2275)",0)
d("UUSEEKER.FOR","RATE","RIN(2276)",0)
d("UUSEEKER.FOR","SNRMIN","RIN(2282)",0)
d("UUSEEKER.FOR","FOVLIM","RIN(2283)",0)
d("UUSEEKER.FOR","SNRACQ","RIN(2284)",0)
d("UUSEEKER.FOR","RFINAL","RIN(2285)",0)
d("UUSEEKER.FOR","ACQRNG","RIN(2286)",0)
d("UUSEEKER.FOR","TRGSIG","RIN(2302)",0)
d("UUSEEKER.FOR","RNGTRK","RIN(2372)",0)
d("UUSEEKER.FOR","RNGTRM","RIN(2374)",0)
d("UUSEEKER.FOR","SEKTYP","IIN(41)",0)
d("UUSEEKER.FOR","ITRGSG","IIN(42)",0)
d("UUSEEKER.FOR","BCKGRD","IIN(43)",0)
d("UUTARGET.FOR","TARLEN","RIN(3660)",0)
d("UUTARGET.FOR","TARWID","RIN(3661)",0)
d("UUTARGET.FOR","GMU","RIN(6)",0)
d("UUTARGET.FOR","TARPOS","RIN(3662)",0)
d("UUTARGET.FOR","TARVEL","RIN(3665)",0)
d("UUTARGET.FOR","TARRI","RIN(3668)",0)
d("UUTARGET.FOR","CSOPOS","RIN(3669)",0)
d("UUTARGET.FOR","CSOVEL","RIN(3672)",0)
d("UUTARGET.FOR","CSORI","RIN(3675)",0)
d("UUTARGET.FOR","TNKPOS","RIN(3676)",0)
d("UUTARGET.FOR","TNKVEL","RIN(3679)",0)
d("UUTARGET.FOR","TNKRI","RIN(3682)",0)
d("UUTARGET.FOR","RHOPOS","RIN(3683)",0)
d("UUTARGET.FOR","RHOVEL","RIN(3686)",0)
d("UUTARGET.FOR","RHORI","RIN(3689)",0)
d("UUTARGET.FOR","CLTPOS","RIN(3690)",0)
d("UUTARGET.FOR","CLTVEL","RIN(3693)",0)
d("UUTARGET.FOR","CLTRI","RIN(3696)",0)
d("UUTARGET.FOR","DTR","RIN(2)",0)
d("UUTARGET.FOR","WIDTH","RIN(3616)",0)
d("UUTARGET.FOR","FOCLEN","RIN(3614)",0)
d("UUTARGET.FOR","RMULT","RIN(3697)",0)
d("UUTARGET.FOR","NOBJ","IIN(260)",0)
d("UUTARGET.FOR","ISKOUT","IIN(241)",0)
d("UUTARGET.FOR","SEKTYP","IIN(41)",0)
d("UUTARGET.FOR","NTARRS","IIN(261)",0)
```

Figure 4.48: EQUIVALENCE example output (EXAMPLE.OUT).

Limitations of the EQUIVALENCE program:

1. The program was designed specifically for the EXOSIM 2.0 program.

4.4.3. COMMON program

The COMMON program takes as input a FORTRAN 77 program in order to extract global variable information from common block statements. The output produced is compatible with the PROLOG program explained later. Refer to Appendix D for the complete program source.

The command line syntax is:

common <input file> <output file>

where:

<input file> = input file name
<output file> = output file name

The following figures will be used to demonstrate the COMMON program.

```
default:        example.out

example.out:    example.f
        common example.f example.out
```

Figure 4.49: COMMON example makefile.

```
      SUBROUTINE ACCEL
C     COMMON "RACCEL" USED FOR MIDFLIGHT CAPABILITIES ONLY

      COMMON / RACCEL / DRSIGA, PSIA   , THTA   , PHIA   , THXZA ,
     .                  THXYA , THYZA  , THYXA  , THZYA  , THZXA ,
     .                  SF1A  , SF2A   , DCA    , TOACCE , GRLST ,
     .                  XYZDP , ABI2   , ABI1   , ABO2   , ABO1
      END

      SUBROUTINE KALMAN
C     COMMON "RKALMN" USED FOR MIDFLIGHT CAPABILITIES ONLY

      COMMON / RKALMN / TKF    , IDRTOK , PP11   , PP12   , PP22   ,
     .                  PY11   , PY12   , PY22   , PLMDFP , YLMDFP ,
     .                  PLAMH  , YLAMH  , PLAMDH , YLAMDH , PLAMDF ,
     .                  YLAMDF , TGIL   , KFMODE , IFPAS
      END
```

Figure 4.50: COMMON example input (EXAMPLE.F).

Using the above figure as input (a FORTRAN 77 program fragment from EXOSIM 2.0), the EQUIVALENCE program produced the following output. The information contained about each variables includes:

1. source file name
2. local variable name
3. common variable name
4. variable usage count

```
d("UUACCEL.FOR","DRSIGA","RACCEL.DRSIGA",0)
d("UUACCEL.FOR","PSIA","RACCEL.PSIA",0)
d("UUACCEL.FOR","THTA","RACCEL.THTA",0)
d("UUACCEL.FOR","PHIA","RACCEL.PHIA",0)
d("UUACCEL.FOR","THXZA","RACCEL.THXZA",0)
d("UUACCEL.FOR","THXYA","RACCEL.THXYA",0)
d("UUACCEL.FOR","THYZA","RACCEL.THYZA",0)
d("UUACCEL.FOR","THYXA","RACCEL.THYXA",0)
d("UUACCEL.FOR","THZYA","RACCEL.THZYA",0)
d("UUACCEL.FOR","THZXA","RACCEL.THZXA",0)
d("UUACCEL.FOR","SF1A","RACCEL.SF1A",0)
d("UUACCEL.FOR","SF2A","RACCEL.SF2A",0)
d("UUACCEL.FOR","DCA","RACCEL.DCA",0)
d("UUACCEL.FOR","T0ACCE","RACCEL.T0ACCE",0)
d("UUACCEL.FOR","GRLST","RACCEL.GRLST",0)
d("UUACCEL.FOR","XYZDP","RACCEL.XYZDP",0)
d("UUACCEL.FOR","ABI2","RACCEL.ABI2",0)
d("UUACCEL.FOR","ABI1","RACCEL.ABI1",0)
d("UUACCEL.FOR","ABO2","RACCEL.ABO2",0)
d("UUACCEL.FOR","ABO1","RACCEL.ABO1",0)
d("UUKALMAN.FOR","TKF","RKALMN.TKF",0)
d("UUKALMAN.FOR","IDRTOK","RKALMN.IDRTOK",0)
d("UUKALMAN.FOR","PP11","RKALMN.PP11",0)
d("UUKALMAN.FOR","PP12","RKALMN.PP12",0)
d("UUKALMAN.FOR","PP22","RKALMN.PP22",0)
d("UUKALMAN.FOR","PY11","RKALMN.PY11",0)
d("UUKALMAN.FOR","PY12","RKALMN.PY12",0)
d("UUKALMAN.FOR","PY22","RKALMN.PY22",0)
d("UUKALMAN.FOR","PLMDFP","RKALMN.PLMDFP",0)
d("UUKALMAN.FOR","YLMDFP","RKALMN.YLMDFP",0)
d("UUKALMAN.FOR","PLAMH","RKALMN.PLAMH",0)
d("UUKALMAN.FOR","YLAMH","RKALMN.YLAMH",0)
d("UUKALMAN.FOR","PLAMDH","RKALMN.PLAMDH",0)
d("UUKALMAN.FOR","YLAMDH","RKALMN.YLAMDH",0)
d("UUKALMAN.FOR","PLAMDF","RKALMN.PLAMDF",0)
d("UUKALMAN.FOR","YLAMDF","RKALMN.YLAMDF",0)
d("UUKALMAN.FOR","TGIL","RKALMN.TGIL",0)
d("UUKALMAN.FOR","KFMODE","RKALMN.KFMODE",0)
d("UUKALMAN.FOR","IFPAS","RKALMN.IFPAS",0)
```

Figure 4.51: COMMON example output (EXAMPLE.OUT).

Limitations of the COMMON program:

1. The program was designed specifically for EXOSIM 2.0 program.

4.4.4. PROLOG utility

The PROLOG program "varusage" was written to assist in the systematic initialization of all required variables. Once other utilities had determined the dependencies and the correct initial values, varusage would then analyze the dependencies and group them into "optimal" sets. This enabled us to combine various types of initializations (BLOCKDATA routines, explicit assignments, and NAMELISTs) into short files of DATA statements which could be included only in the files where they were necessary.

One approach to this is to create a unique include file for each subroutine and main program partition in the multi-partiion application (EXOSIM, in this case). The problem with this is that many of the same initialization statements will appear in multiple include files. Commonly-used variables and constants, like the radius of the earth and π, for example, would have to be initialized in many DATA statements spread across several files. If the initial value of variable

were changed, as in a parametric study, it would be necessary to manually edit many files to make the same change.

Consequently, we chose another approach in which we found subsets of variables which were always used together. For example (hypothetically), we would find that every routine which required an initial value for latitude also required an intial value for longitude (and perhaps other mutually-used values as well). Then, all of these variable names were grouped together in a single include file with the appropriate DATA statements, and a list was maintained of which routines required each particular include file. Of course, there was the possiblity that we would find that the variables did not group particularly well, perhaps resulting in hundreds of include files with only one or two DATA statements in each file. It turned out, though, that many groupings were found, greatly simplifying our file management tasks.

The varusage program was used several times. First, it was used to group the variables found in the many COMMON blocks spread throughout the principle subroutines of EXOSIM. Later, it was applied to the BLOCKDATA variables and to the so-called "dynamic" variables included as COMMON blocks in the main program.

The input to varusage is a list of dependencies in the form

```
d("Filename.src","aliasVARa","VARa",NumRefs)
```

where d is simply a keyword which actually corresponds to a PROLOG predicate, Filename.src is a fortran source file name, aliasVARa is the commonly-used short name of an initialized variable used within Filename.src, VARa is a longer, more fully descriptive name which we use to indicate the usage of the variable within a COMMON block, and NumRefs is the actual number of times that the variable is referenced. This input list is created automatically by a separate utility program which parses the various subroutines (or main program segments, in some cases) and outputs a line for each variable under consideration, even if it is not referenced at all (in which case NumRefs is set to 0).

An example input file of dependencies is given below. The actual initial values of variables need not be provided, since varusage simply creates sets of groupings. Other utility programs are used to merge the output of varusage with the known initial values, thus creating the files of DATA statements, which may be classified. In this example, all of the FORTRAN source files end with the extension .F, as in UUKVAUTO.F. The aliasVARa and VARa parameters may seem redundant, since the VARa parameter (in this example) is always aliasVARa prefixed by the COMMON block name in which it was found. In other applicatiions of varusage, however, we used this feature to detect variables which were in the same position of multiple COMMON blocks, but named differently (and thus aliased). This was important to ensure that the differently-named variables were in fact initialized correctly.

Example input file for EXOSIM subroutine COMMON blocks:

```
d("UUKVAUTO.F","SW17","RKVAUT.SW17",5)
d("UUKVAUTO.F","SW18","RKVAUT.SW18",5)
d("UUKVAUTO.F","SW18P","RKVAUT.SW18P",4)
d("UUKVAUTO.F","SW18Y","RKVAUT.SW18Y",4)
d("UUKVAUTO.F","SW19","RKVAUT.SW19",5)
d("UUKVAUTO.F","SW19P","RKVAUT.SW19P",7)
```

```
d("UUKVAUTO.F","SW19Y","RKVAUT.SW19Y",7)
d("UUKVAUTO.F","IROLL","RKVAUT.IROLL",3)
d("UUKVAUTO.F","TPTON2","RKVAUT.TPTON2",15)
d("UUKVAUTO.F","TYTON2","RKVAUT.TYTON2",15)
d("UUKVAUTO.F","TNEXTP","RKVAUT.TNEXTP",5)
d("UUKVAUTO.F","TNEXTY","RKVAUT.TNEXTY",5)
d("UUKVAUTO.F","FLTCPL","RKVAUT.FLTCPL",2)
d("UUKVAUTO.F","FLTCYL","RKVAUT.FLTCYL",2)
d("UUACCEL.F","DRSIGA","RACCEL.DRSIGA",3)
d("UUACCEL.F","PSIA","RACCEL.PSIA",3)
d("UUACCEL.F","THTA","RACCEL.THTA",3)
d("UUACCEL.F","PHIA","RACCEL.PHIA",3)
d("UUACCEL.F","THXZA","RACCEL.THXZA",2)
d("UUACCEL.F","THXYA","RACCEL.THXYA",2)
d("UUACCEL.F","THYZA","RACCEL.THYZA",2)
d("UUACCEL.F","THYXA","RACCEL.THYXA",2)
d("UUACCEL.F","THZYA","RACCEL.THZYA",2)
d("UUACCEL.F","THZXA","RACCEL.THZXA",2)
d("UUACCEL.F","SF1A","RACCEL.SF1A",6)
d("UUACCEL.F","SF2A","RACCEL.SF2A",6)
d("UUACCEL.F","DCA","RACCEL.DCA",4)
d("UUACCEL.F","T0ACCE","RACCEL.T0ACCE",2)
d("UUACCEL.F","GRLST","RACCEL.GRLST",6)
d("UUACCEL.F","XYZDP","RACCEL.XYZDP",6)
d("UUACCEL.F","ABI2","RACCEL.ABI2",3)
d("UUACCEL.F","ABI1","RACCEL.ABI1",4)
d("UUACCEL.F","ABO2","RACCEL.ABO2",3)
d("UUACCEL.F","ABO1","RACCEL.ABO1",4)
d("UUACSTHR.F","TREFLA","RACSTR.TREFLA",0)
d("UUACSTHR.F","TLSTC","RACSTR.TLSTC",0)
d("UUACSTHR.F","ACSF","RACSTR.ACSF",22)
d("UUACSTHR.F","AOFF1","RACSTR.AOFF1",6)
d("UUACSTHR.F","AOFF2","RACSTR.AOFF2",6)
d("UUACSTHR.F","TMACSA","RACSTR.TMACSA",43)
d("UUACSTHR.F","THACSA","RACSTR.THACSA",39)
d("UUACSTHR.F","LENA","RACSTR.LENA",26)
d("UUACSTHR.F","TMACSB","RACSTR.TMACSB",43)
d("UUACSTHR.F","THACSB","RACSTR.THACSB",39)
d("UUACSTHR.F","LENB","RACSTR.LENB",26)
d("UUAERO.F","TLSTR","RAERO.TLSTR",0)
d("UUAERO.F","MACHL","RAERO.MACHL",0)
d("UUAERO.F","ALFATL","RAERO.ALFATL",0)
d("UUATMOS.F","TLSTA","RATMOS.TLSTA",0)
d("UUATMOS.F","ALTL","RATMOS.ALTL",0)
d("UUBTHRST.F","TLSTB","RBTHRT.TLSTB",0)
d("UUBTHRST.F","T0L","RBTHRT.T0L",0)
d("UUBTHRST.F","BOFF2","RBTHRT.BOFF2",4)
d("UUFRCTHR.F","TLSTF","RFRTHR.TLSTF",0)
d("UUFRCTHR.F","TREFL","RFRTHR.TREFL",0)
d("UUFRCTHR.F","VCOD","RFRTHR.VCOD",0)
d("UUGYRO.F","PSIG","RGYRO.PSIG",3)
d("UUGYRO.F","THTG","RGYRO.THTG",3)
d("UUGYRO.F","PHIG","RGYRO.PHIG",3)
d("UUGYRO.F","THXZG","RGYRO.THXZG",2)
d("UUGYRO.F","THXYG","RGYRO.THXYG",2)
d("UUGYRO.F","THYZG","RGYRO.THYZG",2)
d("UUGYRO.F","THYXG","RGYRO.THYXG",2)
d("UUGYRO.F","THZYG","RGYRO.THZYG",2)
d("UUGYRO.F","THZXG","RGYRO.THZXG",2)
d("UUGYRO.F","SF1G","RGYRO.SF1G",6)
d("UUGYRO.F","SF2G","RGYRO.SF2G",6)
d("UUGYRO.F","DCG","RGYRO.DCG",4)
d("UUGYRO.F","T0GYRO","RGYRO.T0GYRO",2)
d("UUGYRO.F","CIMO","RGYRO.CIMO",2)
d("UUGYRO.F","WBI2","RGYRO.WBI2",3)
d("UUGYRO.F","WBI1","RGYRO.WBI1",4)
d("UUGYRO.F","WBO2","RGYRO.WBO2",3)
d("UUGYRO.F","WBO1","RGYRO.WBO1",4)
d("UUGYRO.F","DRSIGG","RGYRO.DRSIGG",3)
d("UUKALMAN.F","TKF","RKALMN.TKF",4)
d("UUKALMAN.F","IDRTOK","RKALMN.IDRTOK",3)
d("UUKALMAN.F","PP11","RKALMN.PP11",15)
d("UUKALMAN.F","PP12","RKALMN.PP12",8)
d("UUKALMAN.F","PP22","RKALMN.PP22",12)
d("UUKALMAN.F","PY11","RKALMN.PY11",15)
d("UUKALMAN.F","PY12","RKALMN.PY12",8)
d("UUKALMAN.F","PY22","RKALMN.PY22",12)
d("UUKALMAN.F","PLMDFP","RKALMN.PLMDFP",2)
d("UUKALMAN.F","YLMDFP","RKALMN.YLMDFP",2)
d("UUKALMAN.F","PLAMH","RKALMN.PLAMH",4)
d("UUKALMAN.F","YLAMH","RKALMN.YLAMH",4)
```

```
d("UUKALMAN.F","PLAMDH","RKALMN.PLAMDH",6)
d("UUKALMAN.F","YLAMDH","RKALMN.YLAMDH",6)
d("UUKALMAN.F","PLAMDF","RKALMN.PLAMDF",3)
d("UUKALMAN.F","YLAMDF","RKALMN.YLAMDF",3)
d("UUKALMAN.F","KFMODE","RKALMN.KFMODE",17)
d("UUKALMAN.F","IFPAS","RKALMN.IFPAS",5)
d("UUMASSPR.F","TLSTM","RMASS.TLSTM",0)
d("UUMASSPR.F","MASSL","RMASS.MASSL",0)
d("UUMCAUTO.F","ANGACL","RMAUTO.ANGACL",12)
d("UUMCAUTO.F","IMCPAS","RMAUTO.IMCPAS",8)
d("UUMCAUTO.F","TP2END","RMAUTO.TP2END",3)
d("UUMCAUTO.F","TP3END","RMAUTO.TP3END",4)
d("UUMCAUTO.F","IP2END","RMAUTO.IP2END",4)
d("UUMCAUTO.F","TCOAST","RMAUTO.TCOAST",4)
d("UUMCAUTO.F","ICOAST","RMAUTO.ICOAST",4)
d("UUMCAUTO.F","TRDONE","RMAUTO.TRDONE",4)
d("UUMCAUTO.F","IRATE","RMAUTO.IRATE",5)
d("UUMCAUTO.F","IACSB1","RMAUTO.IACSB1",10)
d("UUMCAUTO.F","IACSB2","RMAUTO.IACSB2",3)
d("UUMCAUTO.F","ICNT","RMAUTO.ICNT",10)
d("UUMCAUTO.F","IVPFL","RMAUTO.IVPFL",39)
d("UUMCAUTO.F","IVPFLN","RMAUTO.IVPFLN",7)
d("UUMCAUTO.F","TBURN2","RMAUTO.TBURN2",3)
d("UUMCAUTO.F","OMEGAI","RMAUTO.OMEGAI",3)
d("UUMCAUTO.F","TLSTMA","RMAUTO.TLSTMA",2)
d("UUMCAUTO.F","AACCEL","RMAUTO.AACCEL",20)
d("UUMCGUID.F","ISEQ","RMGUID.ISEQ",12)
d("UUMCGUID.F","TVCOMP","RMGUID.TVCOMP",3)
d("UUMCGUID.F","OMEGA0","RMGUID.OMEGA0",6)
d("UUMCGUID.F","IMIDB2","RMGUID.IMIDB2",3)
d("UUMCGUID.F","TMIDB2","RMGUID.TMIDB2",2)
d("UUMCGUID.F","ISK3ON","RMGUID.ISK3ON",2)
d("UUMISSIL.F","XYZLCH","RMISSL.XYZLCH",6)
d("UUNAVIG.F","GRLAST","RNAVIG.GRLAST",6)
d("UUNAVIG.F","MNAV","RNAVIG.MNAV",0)
d("UUNAVIG.F","DTX0","RNAVIG.DTX0",3)
d("UUNAVIG.F","DTY0","RNAVIG.DTY0",3)
d("UUNAVIG.F","DTZ0","RNAVIG.DTZ0",3)
d("UUNORM.F","GSET","NORCOM.GSET",2)
d("UUNORM.F","ISET","NORCOM.ISET",4)
d("UUOBTARG.F","FIRST2","ROBTRG.FIRST2",2)
d("UUOBTARG.F","GRTPST","ROBTRG.GRTPST",2)
d("UUSSPLAG.F","NLATCH","RSPLAG.NLATCH",46)
d("UUSSPLAG.F","TLATCH","RSPLAG.TLATCH",6)
d("UUSSPLAG.F","LAMMSV","RSPLAG.LAMMSV",10)
d("UUSSPLAG.F","RRELSV","RSPLAG.RRELSV",15)
d("UUSSPLAG.F","VRELSV","RSPLAG.VRELSV",15)
d("UUSSPLAG.F","TI2MSV","RSPLAG.TI2MSV",45)
d("UUSSPLAG.F","SNRSV","RSPLAG.SNRSV",5)
d("UUTARGET.F","TL1","RTARG.TL1",3)
d("UUTARGET.F","GRTLST","RTARG.GRTLST",2)
d("UUTARGET.F","FIRST1","RTARG.FIRST1",2)
d("UUVCSTHR.F","TREFLV","RVCSTR.TREFLV",0)
d("UUVCSTHR.F","TLSTV","RVCSTR.TLSTV",0)
d("UUVCSTHR.F","TMVCS","RVCSTR.TMVCS",17)
d("UUVCSTHR.F","THVCS","RVCSTR.THVCS",13)
d("UUVCSTHR.F","LENVCS","RVCSTR.LENVCS",3)
```

The output of varusage for this example is given below. There are two output files. One lists the merged variables and the other lists the source files which in effect "need" each of the merged groups. First we show the merged variable groups. The first few lines are warning messages, indicating that some variables were not used at all. The significant part of the output follows, headed by the words "Merged lists of dependencies." Within this section are multiple lists, each beginning with a program-generated filename for the DATA statements. This filename is used as a cross-reference to the other output file. This list of depencencies is fed to another utility which actually created the files of DATA statements.

Example output file (1 of 2):

```
RACSTR.TREFLA/TREFLA not used in UUACSTHR.F
RACSTR.TLSTC/TLSTC not used in UUACSTHR.F
RAERO.TLSTR/TLSTR not used in UUAERO.F
```

```
RAERO.MACHL/MACHL not used in UUAERO.F
RAERO.ALFATL/ALFATL not used in UUAERO.F
RATMOS.TLSTA/TLSTA not used in UUATMOS.F
RATMOS.ALTL/ALTL not used in UUATMOS.F
RBTHRT.TLSTB/TLSTB not used in UUBTHRST.F
RBTHRT.T0L/T0L not used in UUBTHRST.F
RFRTHR.TLSTF/TLSTF not used in UUFRCTHR.F
RFRTHR.TREFL/TREFL not used in UUFRCTHR.F
RFRTHR.VCOD/VCOD not used in UUFRCTHR.F
RMASS.TLSTM/TLSTM not used in UUMASSPR.F
RMASS.MASSL/MASSL not used in UUMASSPR.F
RNAVIG.MNAV/MNAV not used in UUNAVIG.F
RVCSTR.TREFLV/TREFLV not used in UUVCSTHR.F
RVCSTR.TLSTV/TLSTV not used in UUVCSTHR.F

Merged lists of dependencies:

^/INCLUDE/SSDYN01.DAT
FLTCYL
FLTCPL
TNEXTY
TNEXTP
TYTON2
TPTON2
IROLL
SW19Y
SW19P
SW19
SW18Y
SW18P
SW18
SW17
%%

^/INCLUDE/SSDYN02.DAT
ABO1
ABO2
ABI1
ABI2
XYZDP
GRLST
T0ACCE
DCA
SF2A
SF1A
THZXA
THZYA
THYXA
THYZA
THXYA
THXZA
PHIA
THTA
PSIA
DRSIGA
%%

^/INCLUDE/SSDYN03.DAT
LENB
THACSB
TMACSB
LENA
THACSA
TMACSA
AOFF2
AOFF1
ACSF
%%

^/INCLUDE/SSDYN04.DAT
BOFF2
%%

^/INCLUDE/SSDYN05.DAT
DRSIGG
WBO1
WBO2
WBI1
WBI2
CIMO
T0GYRO
```

```
DCG
SF2G
SF1G
THZXG
THZYG
THYXG
THYZG
THXYG
THXZG
PHIG
THTG
PSIG
$$

^/INCLUDE/SSDYN06.DAT
IFPAS
KFMODE
YLAMDF
PLAMDF
YLAMDH
PLAMDH
YLAMH
PLAMH
YLMDFP
PLMDFP
PY22
PY12
PY11
PP22
PP12
PP11
IDRTOK
TKF
$$

^/INCLUDE/SSDYN07.DAT
AACCEL
TLSTMA
OMEGAI
TBURN2
IVPFLN
IVPFL
ICNT
IACSB2
IACSB1
IRATE
TRDONE
ICOAST
TCOAST
IP2END
TP3END
TP2END
IMCPAS
ANGACL
$$

^/INCLUDE/SSDYN08.DAT
ISK3ON
TMIDB2
IMIDB2
OMEGA0
TVCOMP
ISEQ
$$

^/INCLUDE/SSDYN09.DAT
XYZLCH
$$

^/INCLUDE/SSDYN10.DAT
DTZ0
DTY0
DTX0
GRLAST
$$

^/INCLUDE/SSDYN11.DAT
ISET
GSET
$$
```

```
^/INCLUDE/SSDYN12.DAT
GRTPST
FIRST2
%%

^/INCLUDE/SSDYN13.DAT
SNRSV
TI2MSV
VRELSV
RRELSV
LAMMSV
TLATCH
NLATCH
%%

^/INCLUDE/SSDYN14.DAT
FIRST1
GRTLST
TL1
%%

^/INCLUDE/SSDYN15.DAT
LENVCS
THVCS
TMVCS
%%
```

The other output file is given below. The peculiar format is actually the macro language of a commonly-used programmer's editor called "Brief." This output file can be compiled and run in the editor to automatically add the necessary include statements back into the source FORTRAN files. Note that the included filenames correspond to those in the list above. For example, the macro below will first open the file UUKVAUTO.F and add the FORTRAN statement

```
$INCLUDE('^/INCLUDE/SSDYN01.DAT)
```

since it knows (from the file above) that ^/INCLUDE/SSDYN01.DAT will initialize several variables that are in fact needed in UUKVAUTO.F. This particular example is not nearly as interesting as some much longer examples, in which the same file of DATA statements is included in more than one source file.

Example output file (2 of 2):

```
(macro insertall
(
(edit_file "UUKVAUTO.F")
(search_fwd "* DATA " 0)
(end_of_line)
(insert "\n$INCLUDE('^/INCLUDE/SSDYN01.DAT')")
(write_buffer)
(edit_file "UUACCEL.F")
(search_fwd "* DATA " 0)
(end_of_line)
(insert "\n$INCLUDE('^/INCLUDE/SSDYN02.DAT')")
(write_buffer)
(edit_file "UUACSTHR.F")
(search_fwd "* DATA " 0)
(end_of_line)
(insert "\n$INCLUDE('^/INCLUDE/SSDYN03.DAT')")
(write_buffer)
(edit_file "UUBTHRST.F")
(search_fwd "* DATA " 0)
(end_of_line)
(insert "\n$INCLUDE('^/INCLUDE/SSDYN04.DAT')")
(write_buffer)
(edit_file "UUGYRO.F")
(search_fwd "* DATA " 0)
(end_of_line)
(insert "\n$INCLUDE('^/INCLUDE/SSDYN05.DAT')")
(write_buffer)
(edit_file "UUKALMAN.F")
(search_fwd "* DATA " 0)
(end_of_line)
```

```
(insert "\n$INCLUDE('^/INCLUDE/SSDYN06.DAT')")
(write_buffer)
(edit_file "UUMCAUTO.F")
(search_fwd "* DATA " 0)
(end_of_line)
(insert "\n$INCLUDE('^/INCLUDE/SSDYN07.DAT')")
(write_buffer)
(edit_file "UUMCGUID.F")
(search_fwd "* DATA " 0)
(end_of_line)
(insert "\n$INCLUDE('^/INCLUDE/SSDYN08.DAT')")
(write_buffer)
(edit_file "UUMISSIL.F")
(search_fwd "* DATA " 0)
(end_of_line)
(insert "\n$INCLUDE('^/INCLUDE/SSDYN09.DAT')")
(write_buffer)
(edit_file "UUNAVIG.F")
(search_fwd "* DATA " 0)
(end_of_line)
(insert "\n$INCLUDE('^/INCLUDE/SSDYN10.DAT')")
(write_buffer)
(edit_file "UUNORM.F")
(search_fwd "* DATA " 0)
(end_of_line)
(insert "\n$INCLUDE('^/INCLUDE/SSDYN11.DAT')")
(write_buffer)
(edit_file "UUOBTARG.F")
(search_fwd "* DATA " 0)
(end_of_line)
(insert "\n$INCLUDE('^/INCLUDE/SSDYN12.DAT')")
(write_buffer)
(edit_file "UUSSPLAG.F")
(search_fwd "* DATA " 0)
(end_of_line)
(insert "\n$INCLUDE('^/INCLUDE/SSDYN13.DAT')")
(write_buffer)
(edit_file "UUTARGET.F")
(search_fwd "* DATA " 0)
(end_of_line)
(insert "\n$INCLUDE('^/INCLUDE/SSDYN14.DAT')")
(write_buffer)
(edit_file "UUVCSTHR.F")
(search_fwd "* DATA " 0)
(end_of_line)
(insert "\n$INCLUDE('^/INCLUDE/SSDYN15.DAT')")
(write_buffer)
)
)
```

The listing of the source code for the varusage utility program follows. It is written in PROLOG, which is not particularly easy to follow if one is not familiar with the language. PROLOG is actually quite intuitive, once one understands the program flow and basic operations, including the binding of variables. No attempt will be made to explain PROLOG here.

varusage program listing (PROLOG source code):

```
/*
   Find variables used in same sets of files, based
   on an input file of the form
    d("File1.src","aliasVARa","VARa",NumRefs)
    d("File2.src","aliasVARk","VARk",NumRefs)
    etc.
*/
trace

domains
   file          = datafile; varFile; batFile
   sourcefile, variable, alias = string
   one_d_a_record = d(sourcefile,alias,variable,integer)
   filelist      = sourcefile*
   variablelist  = variable*
   aliaslist     = alias*
```

```
database - v_record
   v(aliaslist,variable)
database - d_record
   depends(sourcefile, variable)
database - dl_record
   dl(filelist, variable)
database - cdl_record
   cdl(filelist, variablelist)
database - index_record
   index(integer)

predicates
   file_consult(string)
   repfile(file)
   assert_new_v(alias,variable)
   assert_d_and_maybe_v(one_d_a_record)
   build_depend_lists
   write_depend_lists
   merge_same_depends(filelist)
   retract_one_cdl( filelist, variablelist)
   build_combined_depend_lists
   write_list(aliaslist)
   write_combined_depend_lists
   write_one_cdl(filelist, variablelist)
   write_vlist_w_aliases( variablelist )
   write_variable_w_aliases( variable )
   member(variable, variablelist)
   member(alias, aliaslist)
   member(sourcefile, filelist)
   good_sublist(filelist, filelist)
   sublist(filelist, filelist)
   length(filelist, integer)
   union(filelist, filelist, filelist)
   append(variablelist, variablelist, variablelist)
   superset( filelist, variablelist )
   form_filename(integer, string)
   write_flist(filelist, string)
   mergelists
   go

clauses
   member(X,[H|_]) :-
     X = H.
   member(X,[_|T]) :-
     member(X,T).

   assert_new_v(A,V) :-
     /* checks to see if alias prev used */
     /* always fails, and assertion occurs below */
     v(AL,V1),
     member(A,AL),
     V <> V1,
     write("\Warning!  Alias ",A," is used by both ",V, " and ",V1),
     fail.
   assert_new_v(A,V) :-
     v(AL,V),
     member(A,AL),
     !.
   assert_new_v(A,V) :-
     retract(v(AL,V),v_record),
     AL1 = [A | AL],
     write("\nWarning!  Variable ",V," is known by aliases ",AL1),
     assertz(v(AL1,V),v_record),
     !.      /*  There are more v()'s, but we only need this one */
   assert_new_v(A,V) :-
     assertz(v([A],V),v_record).

   assert_d_and_maybe_v(Term) :-
     Term = d(F,A,V,N),
     N = 0,
     write("\n",V,"/",A," not used in ",F),
     !.
   assert_d_and_maybe_v(Term) :-
     Term = d(F,A,V,_),
     ShortTerm = depends(F,V),
     assertz(ShortTerm,d_record),
     assert_new_v(A,V).

   file_consult(FileName) :-
     openread(datafile,FileName),
     readdevice(datafile),
```

```
  repfile(datafile),
  readterm(one_d_a_record, Term),
  assert_d_and_maybe_v(Term),
  fail.
file_consult(_).

repfile(_).
repfile(F) :- not (eof(F)), repfile(F).

build_depend_lists :-
  v(_,V),
  findall(S,depends(_,V),L),
  assertz(dl(L,V),dl_record),
  fail.
build_depend_lists.

retract_one_cdl( FL, VL ) :-
  retract(cdl( FL, VL ), cdl_record),
  !.

merge_same_depends(FL) :-
  dl(FL,V),
  retract(dl(FL,V), dl_record),
  retract_one_cdl( FL, VL ),
  assertz(cdl( FL, [V|VL] ), cdl_record ),
  fail.
merge_same_depends(_).

build_combined_depend_lists :-
  v(_,V),
  dl(FL,V),
  retract(dl(FL,V), dl_record),
  assertz(cdl( FL, [V] ),cdl_record),
  merge_same_depends(FL),
  fail.
build_combined_depend_lists.

write_depend_lists :-
  dl(FL, V),
  writef("\n%10s : ",V),
  write(FL),
  fail.
write_depend_lists.

write_list([]).
write_list([H|T]) :-
  write("\n",H),
  write_list(T).

write_variable_w_aliases(V) :-
  v(AL,V),
  write_list(AL).

write_vlist_w_aliases( [] ).
write_vlist_w_aliases( [H|TVL] ) :-
  write_variable_w_aliases(H),
  write_vlist_w_aliases(TVL).

write_flist([],_).
write_flist([H|T],IncludeName) :-
  write("(edit_file \"",H,"\")\n"),
  write("(search_fwd \"* DATA \" 0)\n"),
  write("(end_of_line)\n"),
  write("(insert \"\\n$INCLUDE('",IncludeName,"')\")\n"),
  write("(write_buffer)\n"),
  write_flist(T,IncludeName).

form_filename(I,FN) :-
  I < 10,
  str_int(S,I),
  concat("^/INCLUDE/SSDYN0",S,TmpStr),
  concat(TmpStr,".DAT",FN),
  !.
form_filename(I,FN) :-
  str_int(S,I),
  concat("^/INCLUDE/SSDYN",S,TmpStr),
  concat(TmpStr,".DAT",FN),
  !.

write_one_cdl(FL,VL) :-
  writedevice(varFile),
```

```
  retract(index(I), index_record),
  !,
  form_filename(I,FName),
  write("\n\n",FName," "),
  I1 = I+1,
  assert(index(I1)),
  write_vlist_w_aliases(VL),
  write("\n%%"),
  writedevice(batFile),
  write_flist(FL,FName).

write_combined_depend_lists :-
  assert(index(1)),
  cdl(FL, VL),
  write_one_cdl(FL, VL),
  fail.
write_combined_depend_lists.

sublist([], _).
sublist([H|L1], L2) :-
  member(H,L2),
  sublist(L1,L2).

length([],0).
length([_|T],X) :-
  length(T,Y),
  X = Y+1.

good_sublist(L, L1) :-
  sublist(L, L1),
  !,
  length(L,X),
  length(L1,X1),
  X > 1,
  X1 > X,
  X1-X < 1.

union([], L2, L2).
union([X|L1], L2, L3) :-
  member(X,L2),
  union( L1, L2, L3 ).
union([X|L1], L2, [X|L3]) :-
  union( L1, L2, L3 ).

append([], L2, L2).
append([X|L1], L2, [X|L3]) :-
  append( L1, L2, L3 ).

superset( FL, VL ) :-
  cdl(FL1, VL1),
  good_sublist(FL, FL1),
  append(VL, VL1, VL2),
  union(FL, FL1, FL2),
  retract(cdl(FL,VL),cdl_record),
  retract(cdl(FL1,VL1),cdl_record),
  assertz(cdl(FL2, VL2),cdl_record),
  !.

mergelists :-
  cdl(FL,VL),
  superset(FL,VL),
  fail.
mergelists.

go :-
  trace(off),
  openwrite(varFile,"VARUSAGE.TXT"),
  openwrite(batFile,"ADDINCL.BAT"),
  writedevice(batFile),
  write("(macro inserta11\n(\n"),
  writedevice(varFile),
  file_consult("exo.txt"),
  build_depend lists,
  build_combined_depend_lists,
  !,
  mergelists,
  trace(off),
  !,
  write("\n\nMerged lists of dependencies:"),
  assertz(v(["!!!"],"!!!")),
  write_combined_depend_lists,
```

```
    writedevice(batFile),
    write(")\n)\n"),
    closefile(batFile),
    closefile(varFile),
    fail.

goal
  go().
```

5. Application Software

During the past contract year, most of the new system software and utilities have been developed to support current application software, primarily EXOSIM 1.0 and EXOSIM 2.0. This chapter describes the EXOSIM activity, as well as some preliminary work with LEAP.

5.1. EXOSIM

EXOSIM is the culmination of a series of exoatmospheric simulations, as shown in Figure 5.1. In this section we will provide a brief overview of the activity which has led up to the current project in which we are attempting to fully parallelize EXOSIM.

ERIS Baseline Specifications — LMSC

KWEST Simulation — BDM
- ACSL/FORTRAN

KEERIS Simulation — CRC (10/88-2/89)
- Boost-phase only
- ACSL/FORTRAN

EXOSIM Version 1.0 Simulation — CRC (3/89-6/89)
- Post-boost, midcourse, KV phases modeled
- All-FORTRAN
- BDM staring FPA seeker

EXOSIM Version 2.0 Simulation — CRC (7/89-10/89)
- Enhanced seeker, IMU
- SP/OP algorithms added
- Modifications to midcourse guidance and attitude control
- All-FORTRAN

Unclassified EXOSIM — Dynetics (1/90-5/90)
- Based on Version 1.0
- First- and second-stage boost only
- Time-driven, not event-driven
- Commented for parallel partitioning (up to five processors)

Parallel EXOSIM 1.0 — Georgia Tech (3/90-present)
- Based on Unclassified EXOSIM
- Boost-phase only
- Partitioned for up to 27 processors
- Ported to high-speed processors
- Demonstrated real-time performance
- Subsequently used for Ada conversion and benchmarks

Parallel EXOSIM 2.0 — Georgia Tech (10/90-present)
- Contains classified data and classified subroutines
- Converted from event-driven to time-driven
- Eliminated all non-portable features
- Midcourse/terminal phase partitioned for 13 processors and demonstrated running in real time
- Comprehensive end-to-end version partitioned for 19 processors and being ported to FPP/FPX processors
- Derivative of end-to-end version used to demonstrate real-time boost-phase only operation

Figure 5.1: Evolution of EXOSIM

5.1.1. EXOSIM 1.0

One of the earlier subcontractors for this work (Dynetics) modified Version 1.0 of EXOSIM, changing it from an event-driven structure to a time-driven structure. At the same time, it was made into an unclassified version by replacing the data set and changing two routines. This modified version of EXOSIM was first implemented at DETL and was described in the annual report for this task in FY 1990. Briefly, we generated a set of guidelines for partitioning FORTRAN code on the PFP and described a means of testing the partitions on a single-processor system. Following these guidelines, Dynetics first produced a first-stage boost version of the modified EXOSIM, partitioned for four processors. This program is called BOOST1. They then produced a first/second-stage boost version (BOOST2), partitioned for five processors. Both of these programs ran correctly on the PFP, requiring only a simple procedure of splitting up the main program along documented partitions and adding the appropriate communication instructions (which is an automated process).

At this time last year (August 1990), we had a 27-processor version of EXOSIM which was essentially a baseline version for the real-time version to be written for the FPPs. This version was modified slightly, removing all but one COMMON block. A version suitable for the Sun-hosted PFP was then developed in several stages. First, in order to reduce the required number of processors without impacting performance, the three center-of-gravity (COG) modules and the three moment-of-inertia modules (MOI) were combined into a single COG module and a single MOI module. Then, in order to accomodate the limited data memory of the PFP, the BAUTO module was split into three modules. This resulted in a 25-processor version.

Since the FPP development relied on the conversion of FORTRAN programs to C, we verified the operation of the FORTRAN-to-C translator. This was accomplished by translating both the single-processor and 25-processor versions of EXOSIM. The resulting C code for the single-processor version was compiled and executed on three machines: the PFP host (running RMX II), a MicroVAX (running Ultrix), and a Sun 4 (running SunOS/Unix). The results were as essentially the same on each system, verifying the conversion performance. The 25-processor C version was tested by using the standard C compilers on the RMX II host. The resulting object code was loaded on the 386 processors in the PFP, and the simulation ran correctly. The single-processor C code is much too large to run on a single FPP, but the 25-processor C code was given to the FPP software development group so that they could test their single-precision and double-precision compilers.

One additional processor was added to the partitioning, creating a 26-processor version that will henceforth be described as FPP-BOOST2. This program was modified 16 more times during an iterative process of developing FORTRAN code on the PFP, verifying correct operation, porting to C on the FPP PFP, and identifying necessary changes (in order to get it to operate on the

FPP's). By always making the changes in the FORTRAN version, we guaranteed that operation of the baseline version is tracked. The FORTRAN code was transferred to a Sun machine, where the FORTRAN-to-C conversion program resides. After the C code was generated, it was transferred back to the RMX-based PFP host and run on the PFP in order to make certain that no errors were introduced during the translation. At this point, we had a portable C version, suitable for the FPPs, so the code was transferred to the Sun host. Since the FPP linker does not support the concept of libraries, all dependencies were explicitly identified. This was done by processing the map files created by the RMXII binder program, which is equivalent to a linker. The map files are run through an "awk" filter, generating a makefile which is ready for execution on the Sun. The process is described in Figure 5.2.

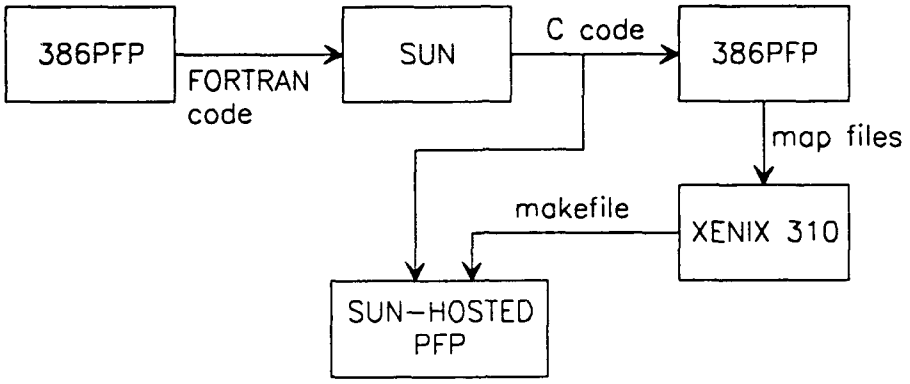


Figure 5.2: Process of porting Parallel EXOSIM 1.0 to a PFP with FPP boards.

In order to assist the FPP group in debugging their code, a data file was created of all of the crossbar communication in a successful run on the RMX-based PFP. To some extent, this data can be used to verify the operation of individual FPPs in the Sun-hosted PFP.

DETL has continued to use EXOSIM 1.0 as a benchmark and as a means of beginning the transition to Ada. As an exercise in the practical application of Ada to parallel simulations, the FORTRAN code was converted to Ada manually. An Ada-to-C translator was used to generate code which could be compiled for the 80386 processors and FPPs.

5.1.2. EXOSIM 2.0

EXOSIM 2.0 has been transferred, in both classified and unclassified versions, to the PFP host, running under RMX II. The unclassified version is simply a sanitized version of the classified version, with no attempt to make substitutions for the classified code and data. Because of this, the unclassified version will not run and is suitable only for compatibility testing (new compilers). Extensive effort was required to make the single-processor version portable among several systems, including VAX VMS, VAX Ultrix, Intel RMX II, and (later) the Inmos Transputer development system. Most of the portability issues arose from using VAX extensions to FORTRAN, rather than adhering strictly to the ANSI FORTRAN-77 specification.

Several non-standard FORTRAN statements were converted. The most significant of these incompatibilities were the NAMELIST statements, now converted to simple assignment statements through an automated process. We have also changed local COMMON blocks to

local static variables (using SAVE statements) and global COMMON blocks to BLOCKDATA routines. Also, several built-in functions were changed from VAX naming conventions to standard FORTRAN-77 names. At the same time, it was necessary to implement certain functions or subroutines which were not available on the RMX II system, including RAN(), DREAL(), and DIMAG().

These changes resulted in classified and unclassified versions which compiled and linked correctly under RMX II. The classified version runs much as it does on the MicroVAX, but deviations are noticed, beginning around the second-stage separation. We had to make some corrections to variables which were not initialized correctly. As a validity check for all of the converted code, we ported it back to the MicroVAX, where it ran correctly.

An additional version of the classified code was also created on the PFP host. In this version, all I/O is done with standard PFP Host I/O routines, enabling the code to be run on a single target processor within the PFP. This program behaves just like the host version, as would be expected. This target version was eventually used as the starting point for partitioning code among multiple processors.

The hardware I/O structure of the FPP board was not compatible with the original Host I/O routines (designed for 286 and 386 processor boards). Consequently, they were rewritten in a manner which allows them to be used with either FPPs or 286/386 boards. This facilitated the porting of code between these different target processors.

Seeker model 3 was not required for the PFP version of EXOSIM2. Consequently, we commented out of the mainline the significant subroutines and variables used by Seeker model 3. After these and other minor changes to remove unnecessary variables, the program size dropped to less than 1MB. The program ran on the PFP to completion but failed to hit the target in the same manner as the host program had earlier. Some errors were identified in NAMELIST assignments, and the corrected version was compiled for both the VMS and RMX systems. The simulation resulted in a similar miss distance for both versions. The altitude and timing of some events were not identical but the simulation seemed to perform correctly. These minor differences have been attributed to the cumulative effects of slight differences in precision between systems, and perhaps to different ordering of operations (which is internal to each compiler).

We modified and recompiled the iRMXII host version of EXOSIM2 to support file 6 (terminal output) and file 51 (file output). The simulation ran to completion and produced the correct answers.

The changes made to date to EXOSIM V2.0 were so numerous that we needed a more exact way to track the changes. We started with the original version (v00) from the VMS system and redid all the changes creating 11 new versions (v01 thru v11) of EXOSIM V2.0. Each version fixes one or more related problems to the program. All of these versions are strictly for a single-processor system.

In order to prevent each partition from having to run a large initialization program (DATIN), we converted the appropriate initializations to DATA statements. We then needed to settle on a convenient grouping of these statements into files which can be selectively included with various routines. (If we put all of the DATA statements with each routine, we would have the same problem we started with -- large memory requirements -- which would cause serious problems when we eventually use the FPP.) We has wrote a PROLOG program which reads in all variable dependencies, identifies shared dependencies (variables which are used by more than one file), and combines identical dependencies (producing groups of variables which are used by exactly the same file or group of files). By running this program on the EXOSIM dependencies, we were able to determine if there is a need to combine even more dependencies (based on subsets, as opposed to simply identical sets).

The PROLOG program "varusage" was run on the actual data dependencies of EXOSIM 2.0. This identified 83 distinct sets of pre-initialized variables which are used by one or more routines. The number of variables in each set ranged from 1 to 48 (some of these are actually arrays), and some sets are used in as many as 8 files. These sets of dependencies were reduced from an original list of over 560 single dependencies. We then looked at the possibilities of merging variable sets when one set is a "close" superset of another. By "close" we mean that the larger set cannot have an excessive number of variables which are not in the smaller set, since each extra variable is needlessly initialized in one or more routines. When we defined "close" as being only one extra variable, "varusage" reduced the 83 sets down to 66. Additional relaxation of the number of extra variables resulted in much smaller benefits, down to about 58 sets when four extra variables were allowed. Although the reduction from 83 to 66 is significant, we opted to stay with the 83 sets, knowing that the code would be as efficient as possible.

A side effect of the work on the equivalence grouping pointed out two program bugs that were in the original FORTRAN from Coleman. The final result is slightly different that the original with a better miss distance. The next version (v15) incorporated the changes in data initialization. Include statements (referencing files with DATA statements) were inserted in place of the equivalences and call to DATIN. We found that this version did not run correctly because a few variables had been modified by scaling factors in DATIN, and some variables were being initialized outside of NAMELISTs which were EQUIVALENCEd -- the so-called "DYNAM" variables. Both of these issues were corrected, and the program then ran without any DATIN calls (using included files of DATA statements instead). We also performed an analysis of the usage of each of the NAMELIST variables to see how many times each was referenced, if at all. We then passed this information to the PROLOG "varusage" program, which was modified to throw out the unused variables before combining them into subsets. This reduced the number of include files down to 70 (from 83). This version (v18 for the host, v18.pfp for a single target processor) became the basis for our partitioning efforts.

We ran our utility program to check for variables which were referenced before they were defined and found over thirty such variables. These were then initialized to zero, which was the apparent intent of the original programmers, in order to make the code more portable.

This same version was converted to an unclassified form by removing all data, as well as the classified routines. It was then modified to make it suitable for the FORTRAN-to-C translator and thus the FPP compiler. In summary, the v18 version of EXOSIM2 was tested and validated on the Ultrix machine, the iRMXII host, and a PFP 386 processor. We translated the unclassified FORTRAN (v18) into C and then compiled all but the mainline. It was then clear, from the code and data sizes, that getting the simulation to fit on the FPP/FPX would be a major problem unless we converted much of it to single precision.

Later, the single-processor version was benchmarked on four processors: an Intel 286 (on an 8 MHz 286/12 board), an Intel 386 (on a 20 MHz 386/12 board), an Inmos T800 (using the 20 MHz SSE host), and a DEC MicroVAX II. The 386 and the T800 were almost twice as fast as the MicroVAX and over four times faster than the 286. Code sizes were similar, with the T800 being slightly larger than the rest, and simulation results (as measured by miss distance) were similar, but not identical, as has been noted before.

After some attempts at running a two-partition version of v18, Richard Pitts and Philip Bingham began to concentrate on partitioning only the midcourse/terminal portion of EXOSIM. This post-boost version can only start from a specific set of post-boost data values. Steve Wachtel and Tom Collins continued to partition an end-to-end simulation.

The general strategy for partitioning EXOSIM is given in Figure 5.3. This illustrates the major functional elements of the simulation, which are capable of running in parallel most of the time. At a higher level, this could be viewed as three main functional elements: the environment, the target, and the interceptor, where the interceptor is composed of four sub-blocks. These six blocks are further subdivided to extract enough parallelism for the system to run in real time.

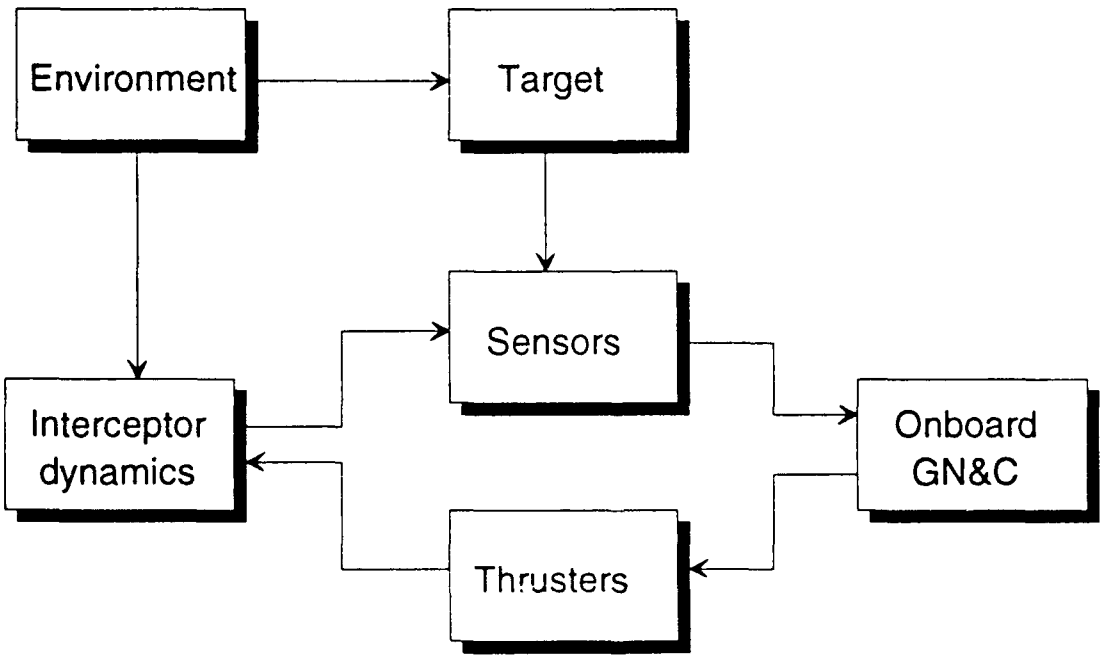


Figure 5.3: General partitioning strategy for EXOSIM 2.0

The following sections describe the various partitioning stages of the end-to-end version of EXOSIM 2.0, beginning with the single-processor version SSV18.. Each step resulted in a new version, named SSVxx.yyz, where "xx" is the major version number, yy is the number of processors (partitions), and z is a letter to distinguish between multiple versions using the same number of processors, but differing in some other aspect..

5.1.2.1. SSV19.3

The two-partition version developed by Richard Pitts and Philip Bingham was investigated, but we learned that it would not run to completion even if the partitions were run serially. Consequently, we started over from the single-processor (v18) version, keeping the partitioned code as close as possible to the original. It made the most sense for the initial split to be between the "truth" model (modelling the physical dynamics of the vehicle and target) and everything else (basically the sensor processing, guidance, navigation, and control).

Our partitioning was along these lines, with system dynamics ("truth states") on one processor and sensors, guidance, navigation, and control on the other. We also split out the output function into a third partition. The main difference between this partition and the earlier version was that we did not split up VCSTHR and ACSTHR, and we did not alter FRACS or FRCTHR. We examined the usage of the variables which were used in both partitions (this is what made Philip Bingham decide to alter his routines), and we determined that most of these variables are really controlled by the GN&C partition (as they should be). The dynamics partition simply modifies some of the variables for local use. A less significant difference was that we placed the GYRO and ACCEL routines in the GN&C partition. These could easily go either way, since sensors can just as easily be considered as part of the dynamics or of the GN&C system, but the structure of the single-processor version favors the chosen placement of these routines.

In many of our earlier simulations, we scheduled all interprocessor communication either at the beginning or the end of the integration timestep. This is consistent with a programming model where only state variables (or only derivatives of state variables) must be communicated. When there are many intermediate variables calculated by algebraic means or by tables, as in the case of EXOSIM, this strategy results in the communcation of variables at inappropriate times, introducing artificial delays and requiring false initialization. This had been attempted by Pitts and Bingham and was probably the main reason that they were unable to get two partitions running in parallel. We began adhering strictly to sending values at the correct times, even though this resulted in the communications being spread throughout the code. The only real problem with this is that it prevented us from using the utility programs which automatically generated the sends and receives, but it provided greater promise of speedup while still retaining the fidelity of the original single-processor version.

This new three-processor partition was designated as "SSV19.3". After creating all required communication calls, we compiled and ran this version. It ran to completion with no errors. SSV19.3 runs about 15% faster than its predecessor, due probably to a better load balance (movement of GYRO and ACCEL) and to the addition of the small third partition.

5.1.2.2. SSV19.5

We then developed a five-partition version. During this same period, we began developing some additional tools to aid in the automatic generation of crossbar code, which we tried on intermediate versions of EXOSIM. With EXOSIM now divided into several partitions of clear physical significance (missile states, target/relative states, IMU, GN&C, and output), we began to concentrate on speedup. Staying with five-processor versions, we first made good use of the first-order (Euler) estimates of the state variables which are calculated at the beginning of each missile state update section. These were sent instead of the values calculated at the end of the loop. The only difference here is the degree of approximation (Euler vs. trapezoidal) and the derivative estimates. This version ran fine, actually improving on the miss distance, which by

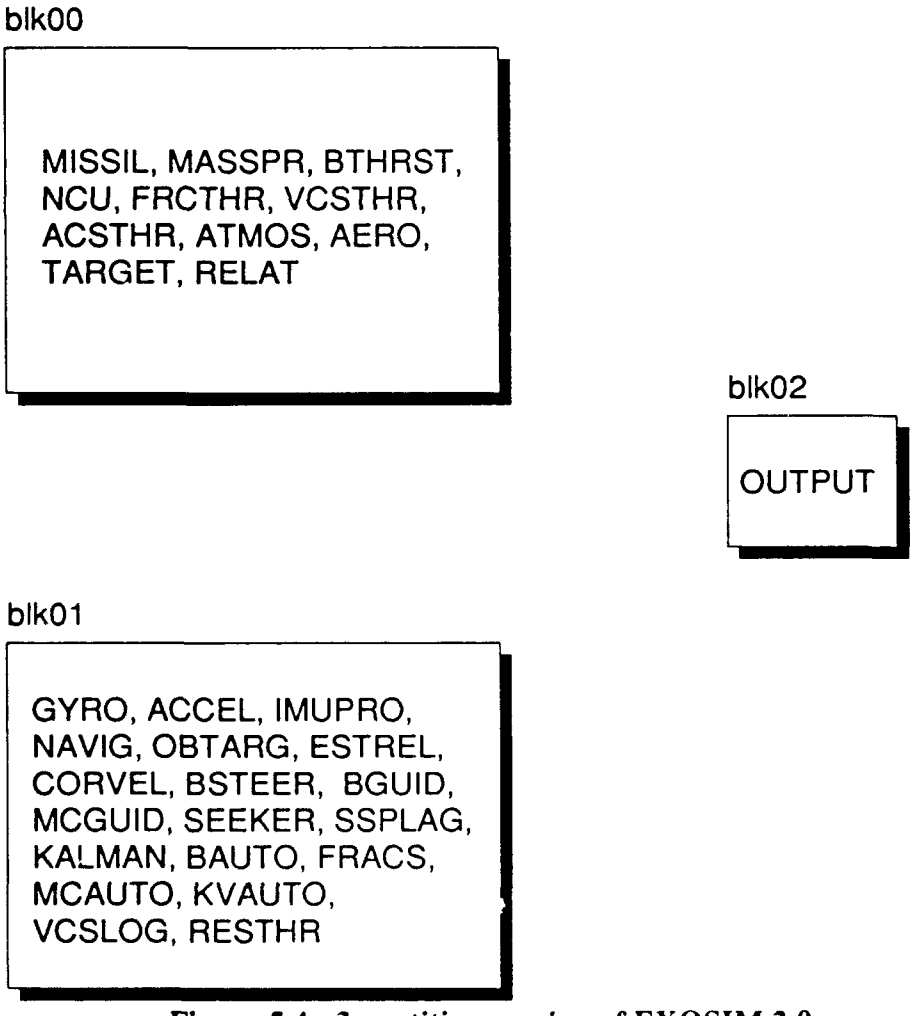


Figure 5.4: 3-partition version of EXOSIM 2.0

itself is only a rough indication of performance. There was about a 10% speedup due to this change.

We then turned to what we consider to be an error in EXOSIM. It does not make sense to perform any type of integration at the beginning of the loop, in either a serial or parallel version. The present-state information should be adequate for the calculation of all derived variables, including state derivatives. We deleted the first-order estimates and began sending the most-recently-calculated state variables at the beginning of each missile state update. This version is about 30% faster than the previous version, at least during first-stage boost, and miss distance was again reduced (by more than half). This improvement could be due to the fact that the GN&C routines are now receiving the present states (as they should), rather than some extrapolated next-states.

As an aside, we uncovered three more errors in the original EXOSIM code. Three variables (TL2, TONAV, and TGIL) are used in the mainline (either as direct assignments or formal parameters) and are also used within COMMON blocks within subroutines. This sort of hidden interaction was supposedly eliminated in EXOSIM 2.0.

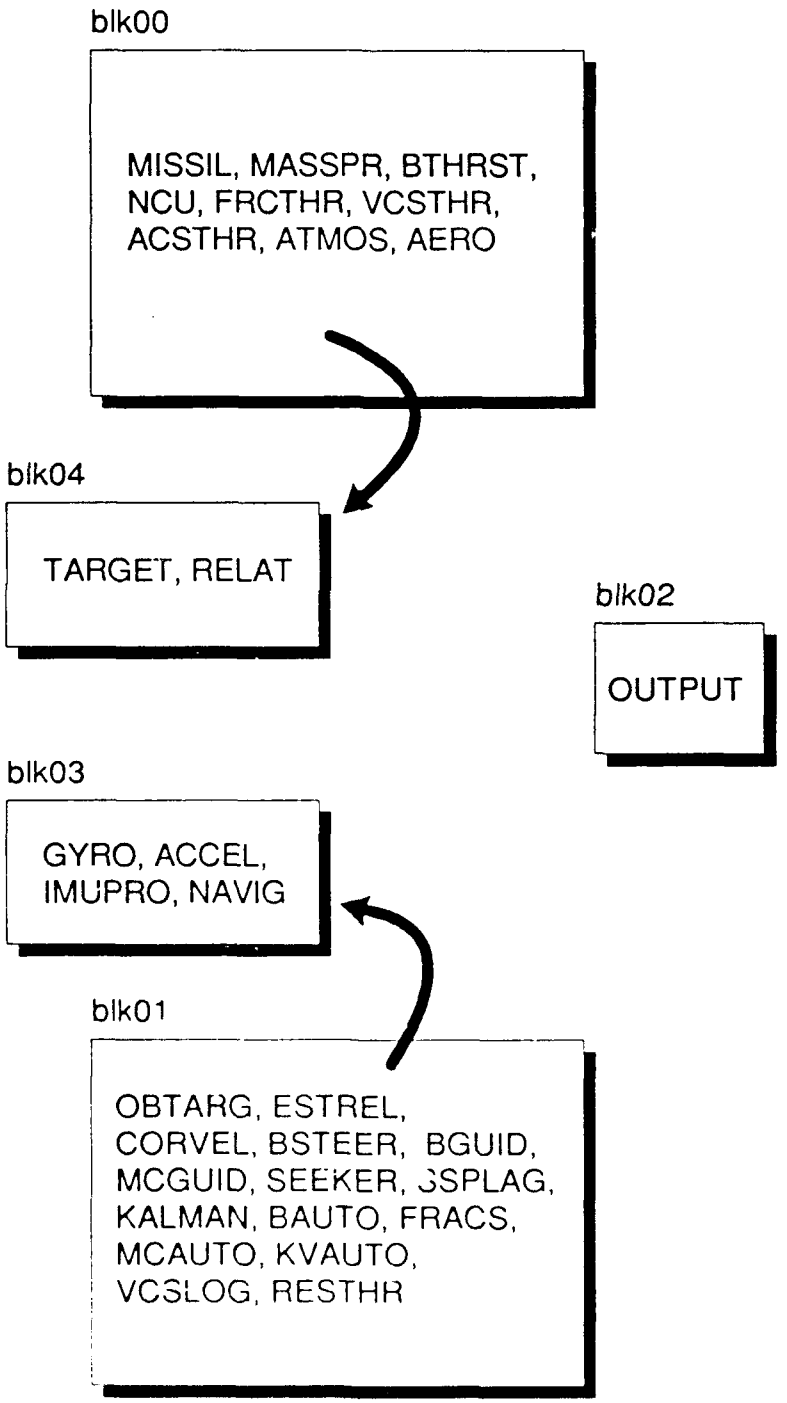


Figure 5.5: 5-partition version of EXOSIM 2.0

5.1.2.3. SSV19.6

We began to focus on the second-stage boost, isolating the computational slowdown to BAUTO. As a rough estimate, BAUTO was about 4-5 times slower than any other candidate partition, and it showed little promise for being split up. (Actually, it made sense for it to exist in the same partition(s) as MCAUTO and KVAUTO, since these routines substitute for each other as time passes.) The problems do not arise from table lookups, as may have been a problem with EXOSIM 1.0, but are due to the calculation of optimal gains, which requires that a discrete model of the missile dynamics be derived, followed by computation of the eigenvalues and eigenvectors (with a great deal of matrix manipulation along the way). Also at this time, we began to perform an extensive timing analysis of the single-processor version.

We then split out a single autopilot partition (for all three autopilot routines), resulting in a six-processor version whose output is identical to the five-processor version, since no reordering of computation had taken place. We had originally planned to improve the speed of BAUTO and perhaps reorder some communications, but as we looked at the optimal gain calculations in BAUTO, we began to doubt their feasibility, not only for real-time simulation, but also for flight. In addition to taking a very long time on the average, there is the possibility that in isolated cases, the routines may not arrive at a solution at all. Perhaps we could spend some time trying to improve the control algorithms, but we deferred this until we have the rest of the simulation running real-time. For this reason, we took a shortcut around the BAUTO problem. Noting that the variation of the plant model (and therefore the optimal gains) depends mostly on atmospheric properties and the changing missile mass, we attempted to fit curves to each of the three optimal gains as a function of altitude. Using Mathematica and Excel, we arrived at two second-order polynomials and an exponential/second-order polynomial for the three gains.

The gains could then be computed very quickly, and the missile performance was nearly identical as far as miss distance was concerned. (The second-stage boost ended at slightly different spatial coordinates, but not enough to make any significant difference in the remainder of the flight.) This somewhat alleviated the need to reorder communications or "cheat" in any way to accomodate BAUTO, since the critical path shifted to other routines.

We began to use Gannt charts from Microsoft Project by manually inserting timing routines in the code, then entering the results into Project manually. We then began to automate this process so that we could analyze the critical path with each successive partition. We determined that it was possible to use the file import features built into Project. We needed a preprocessor that

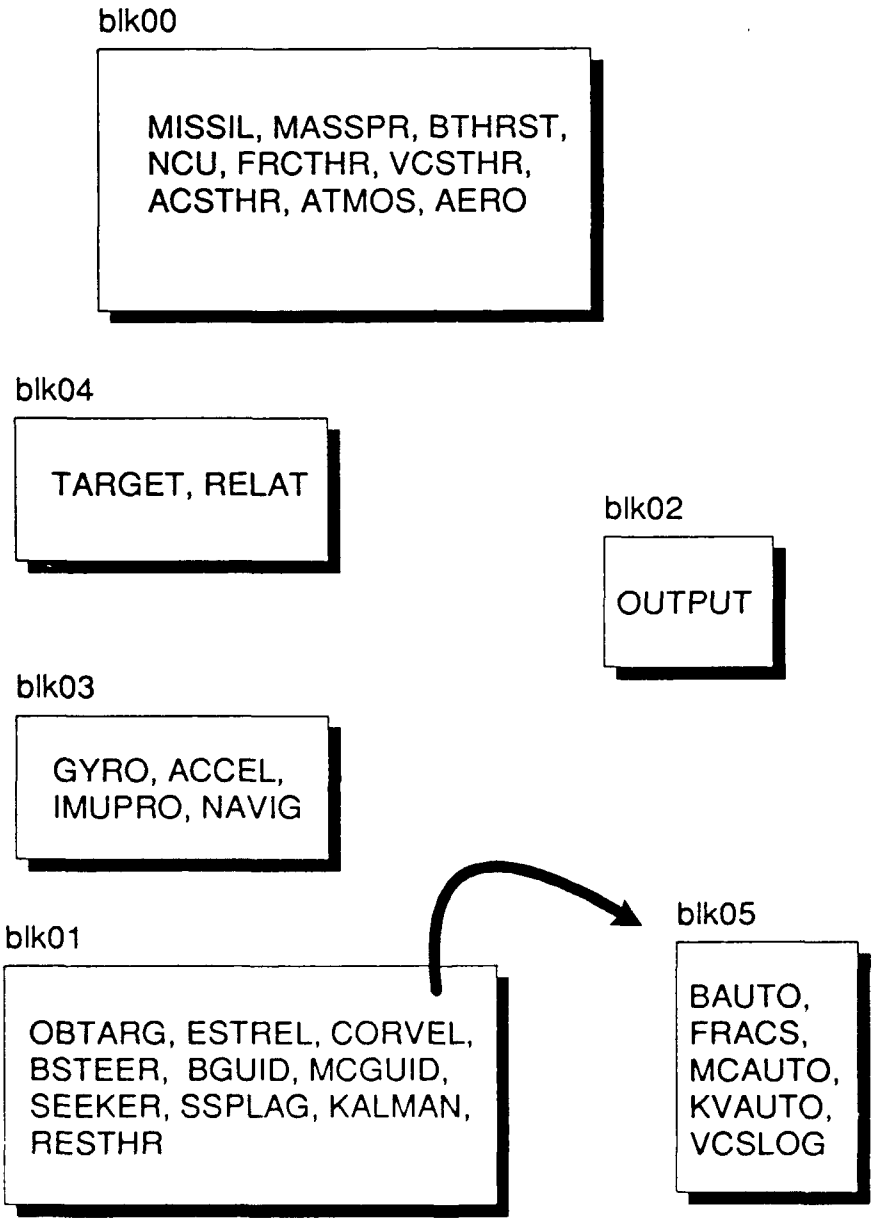


Figure 5.6: 6-partition version of EXOSIM 2.0

analyzed the FORTRAN blocks, inserted timing routines, and kept track of all computation and communication segments, as well as a postprocessor that merged the PFP output with the dependency information. All computation blocks and send routines are considered as distinct events, while receives show up as dependencies (i.e., a computation cannot take place until a receive takes place). We were able to successfully generate the Project information for SSV19.6, but we continued to refine the format and methodology as we proceeded to partition EXOSIM. Some of the timing charts are presented later, beginning with version SSV20.10A.

5.1.2.4. SSV19.7 and SSV19.8

Our seventh block was created by pulling out the atmosphere-related calculations, including ATMOS, AERO, and the computation of altitude and related coordinates. Our eighth block was created by pulling out all of the thrusters (for all phases of flight). Up to this point we were still able to run all routines in the correct order while achieving some significant overlap, but we felt that we should begin to use estimated values for slowly-varying variables when it allowed for more parallelism. We made another eight-processor version which used approximate values for QA, MACH, and PRESS and found no significant change in the output, but this allowed the thruster models to run sooner. We made gradual timing improvements and broke the 10-times-real-time milestone, at least for first-stage boost (where our most recent efforts had focused).

5.1.2.5. SSV20.8

We developed a new major version, SSV20.8, an eight-processor version which had most COMMON blocks removed. We experimented with different levels of optimization and the use of 387 instructions and found that we were able to use the latest Intel FORTRAN compiler at its highest level of optimization, but we are only able to reliably use the 287 floating-point instructions (not the faster 387 instructions). We then converted most integer calculations to two-byte integers, which saved some time without adding another partition. These and other related changes resulted in versions SSV20.8a, SSV20.8b, SSV20.8c, and SSV20.8d.

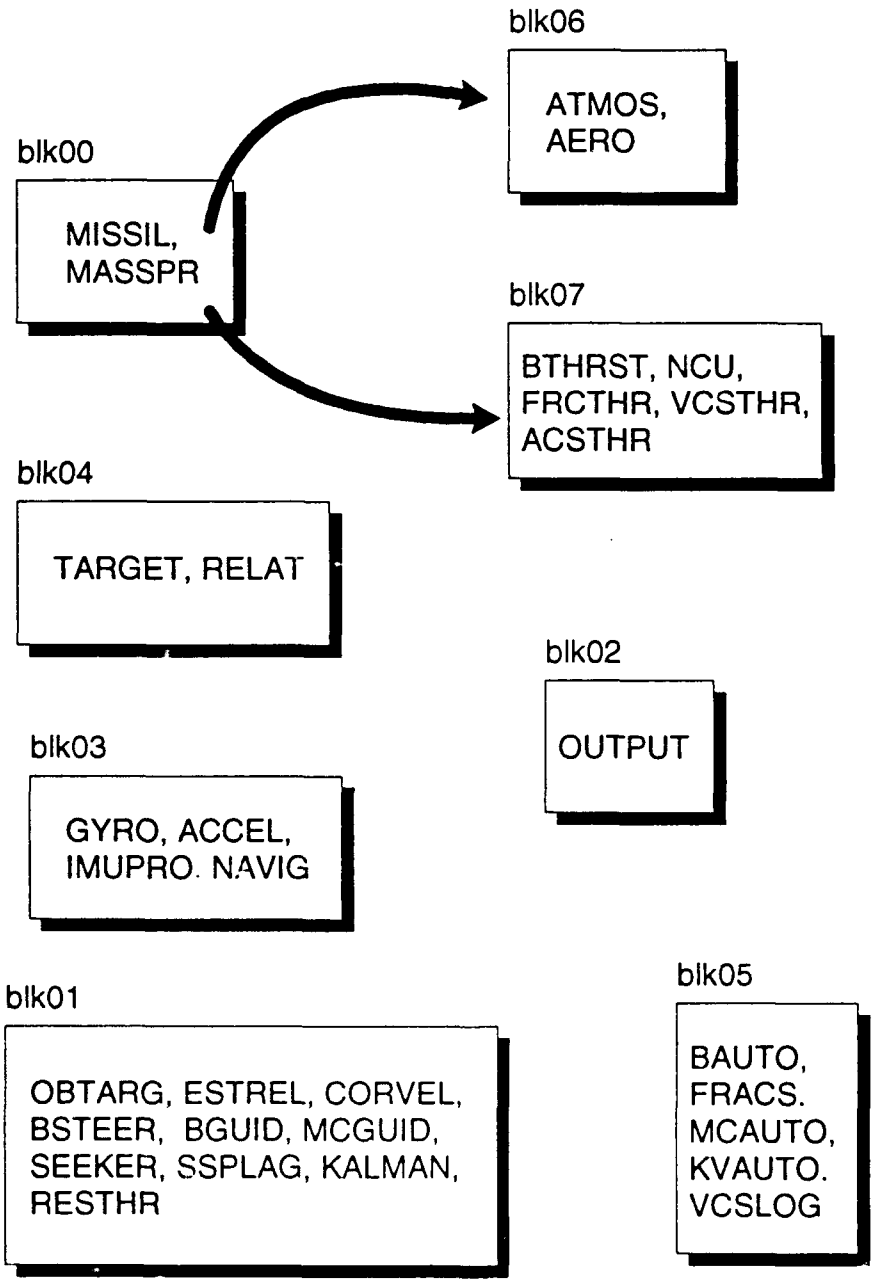


Figure 5.7: 8-partition version of EXOSIM 2.0

5.1.2.6. SSV20.9

We created a nine-processor version, SSV20.9a, by splitting MISSIL into two routines, one for the translational dynamics and one for the rotational dynamics. This worked very well, shifting about 40% of the MISSIL calculations out of the critical path. We continued to use Microsoft Project for the critical path analysis and made some refinements to the dependency information.

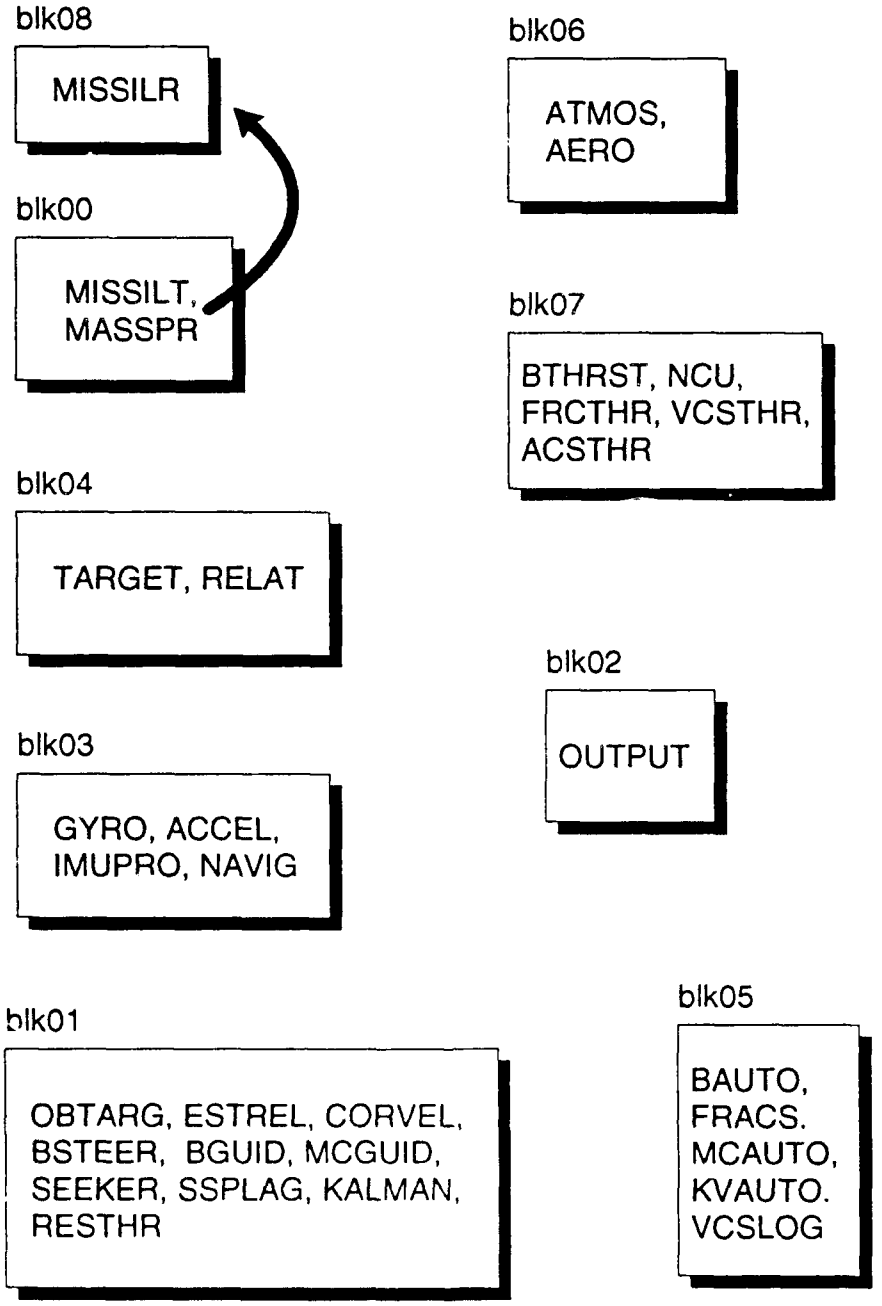


Figure 5.8: 9-partition version of EXOSIM 2.0

EXOSIM 2.0 V20.10A (Stage 1)

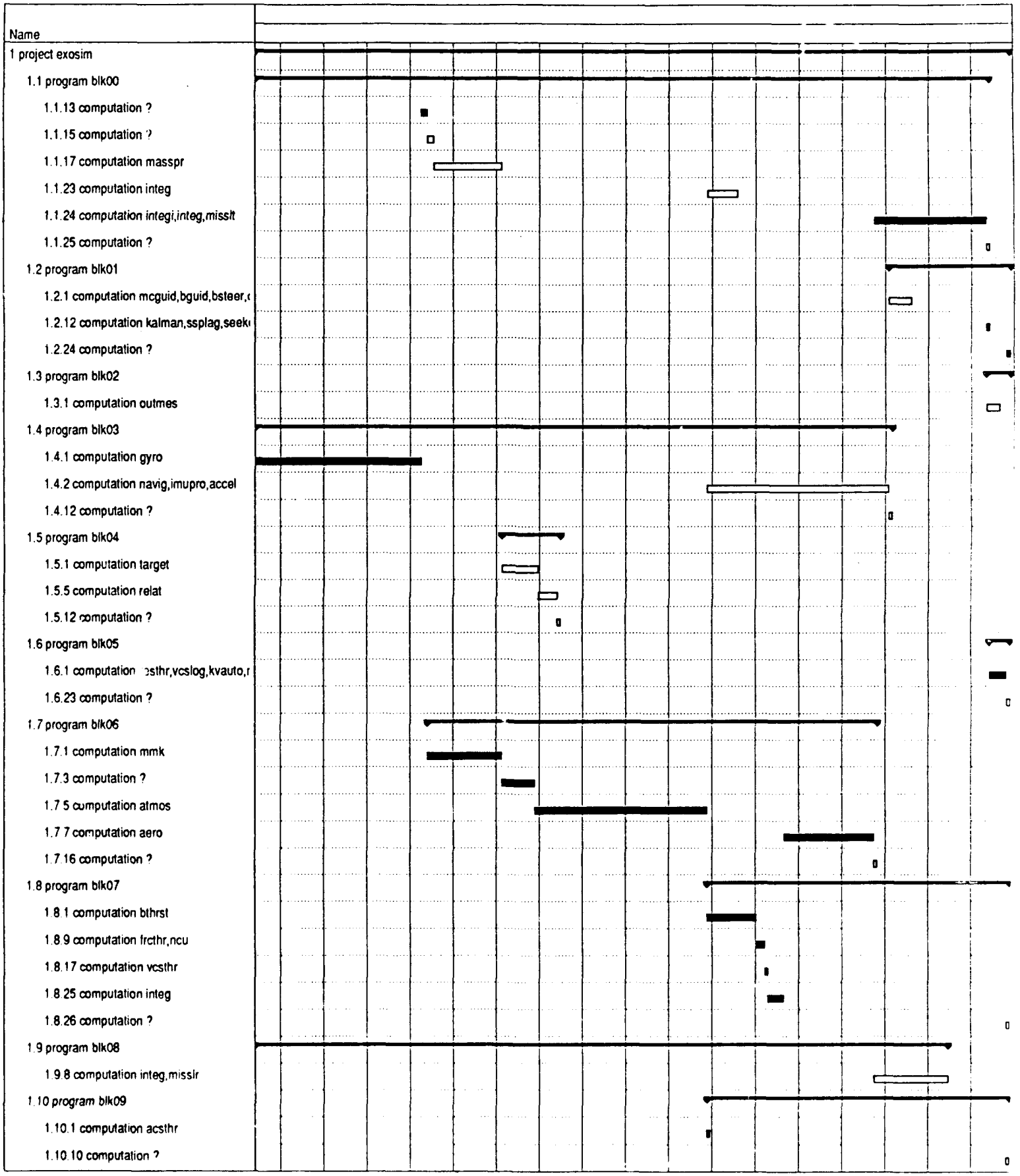


Figure 5.9

5.1.2.7. SSV20.10

We then created a ten-processor version, SSV20.10a, by separating VCSTHR and ACSTHR. These have no cross-dependencies, and VCSTHR was thus removed from the critical path. We also improved the efficiency of MMK by rewriting the ROTMX subroutine.

The timing of this version is illustrated in Figure 5.9. This chart (and all of the ones which will follow) shows the timing of a single integration step averaged over a particular phase of flight. This particular timing chart is for stage 1 (boost phase), so it does not clearly show the motivation for the tenth partition, since both ACSTHR and VCSTHR are not very time-consuming during this portion of flight. It does clearly show the effects of many of the earlier partitions, including the ninth partition, in which the rotational missile states (MISSLR) were split from the translational states (MISSLT). Note that MISSLR and MISSLT run concurrently. The critical path in this timing chart is shown as a solid bar, containing GYRO, MMK, ATMOS, BTHRST, FRCTHR, NCU, VCSTHR, INTEG, AERO, MISSILT, and the autopilot routines. This indicates what areas can be targeted in subsequent partitioning, and we chose to work on the long ATMOS computation.

5.1.2.8. SSV20.11

We created an eleven-processor version by splitting ATMOS into two routines, one which performs four table lookups and one which performs just two lookups, followed by some extensive computation. This was done without changing the results in any way, since the new partition was truly capable of running concurrently. The timing of this version is shown in

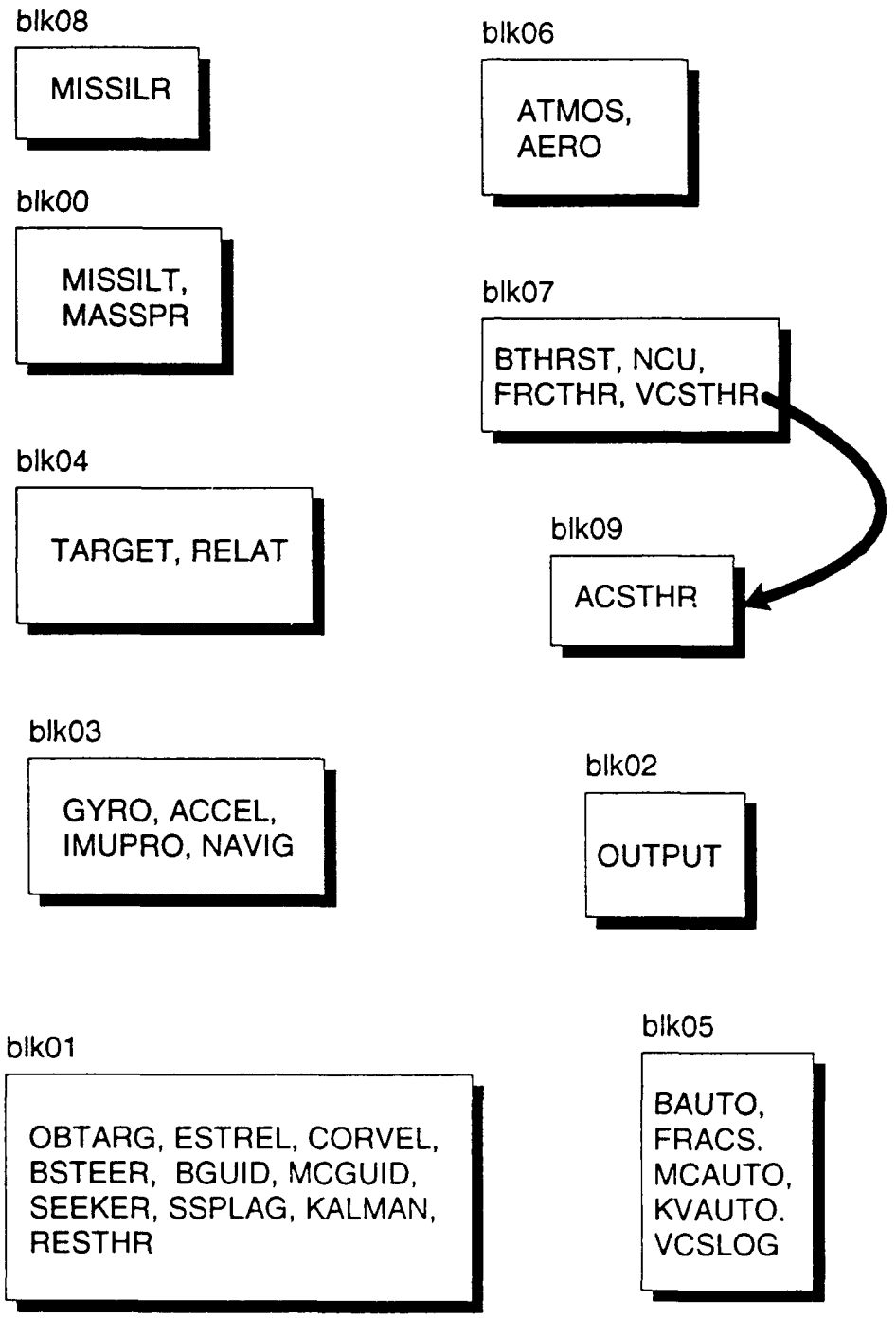


Figure 5.10: 10-partition version of EXOSIM 2.0

Figure 5.12. Again, this timing is for the first stage of flight, which was our primary focus at the time, although we analyzed each of the three major stages (boost, midcourse, and terminal). Note that the ATMOS1 computation begins at the same time as ATMOS2 and that the relative length of ATMOS is thus reduced from the previous version.

This and all of the remaining timing diagrams presented here include duration figures. The basic unit is abbreviated "m" for minute, but this is a misleading carryover from the project-planning software, which provides for no smaller units of time. Actually, an "m" is a tick of our 286/12 or 386/12 boards' onboard timers, and should be thought of simply as an indication of relative time. The timing is described hierarchically, with thinner bars which span the various computations required to implement each partition, or "program." These thinner bars have durations given in elapsed minutes, or "em," which are really no different than regular "minutes" (timer ticks), except that they include idle time during which a program is waiting for data. Time is also allowed for communication of values, but this is not shown on the timing charts, since it makes them very large and does not provide much useful information. Some of the apparent idle time, however, is actually due to communication which has been filtered out of the charts.

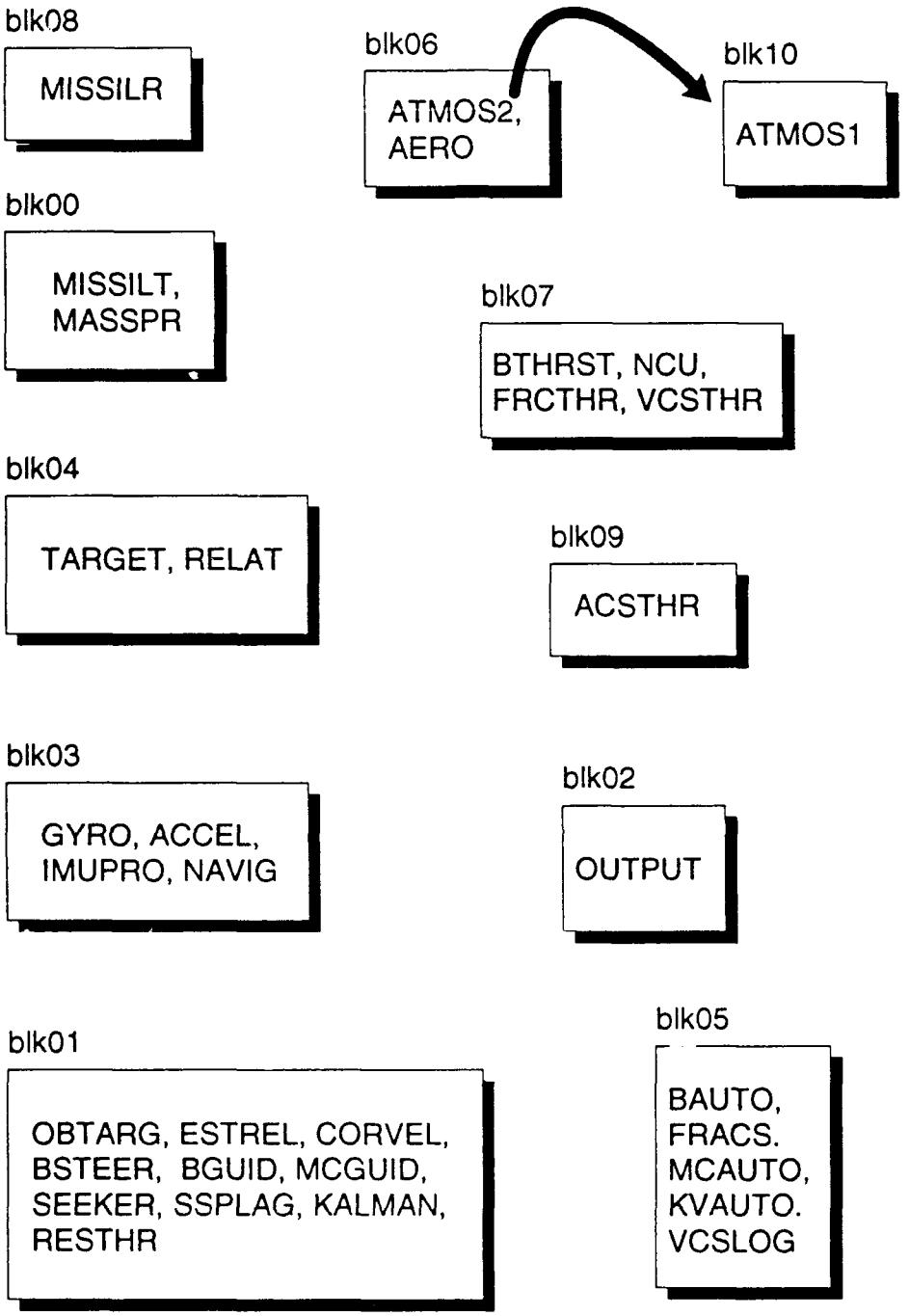


Figure 5.11: 11-partition version of EXOSIM 2.0

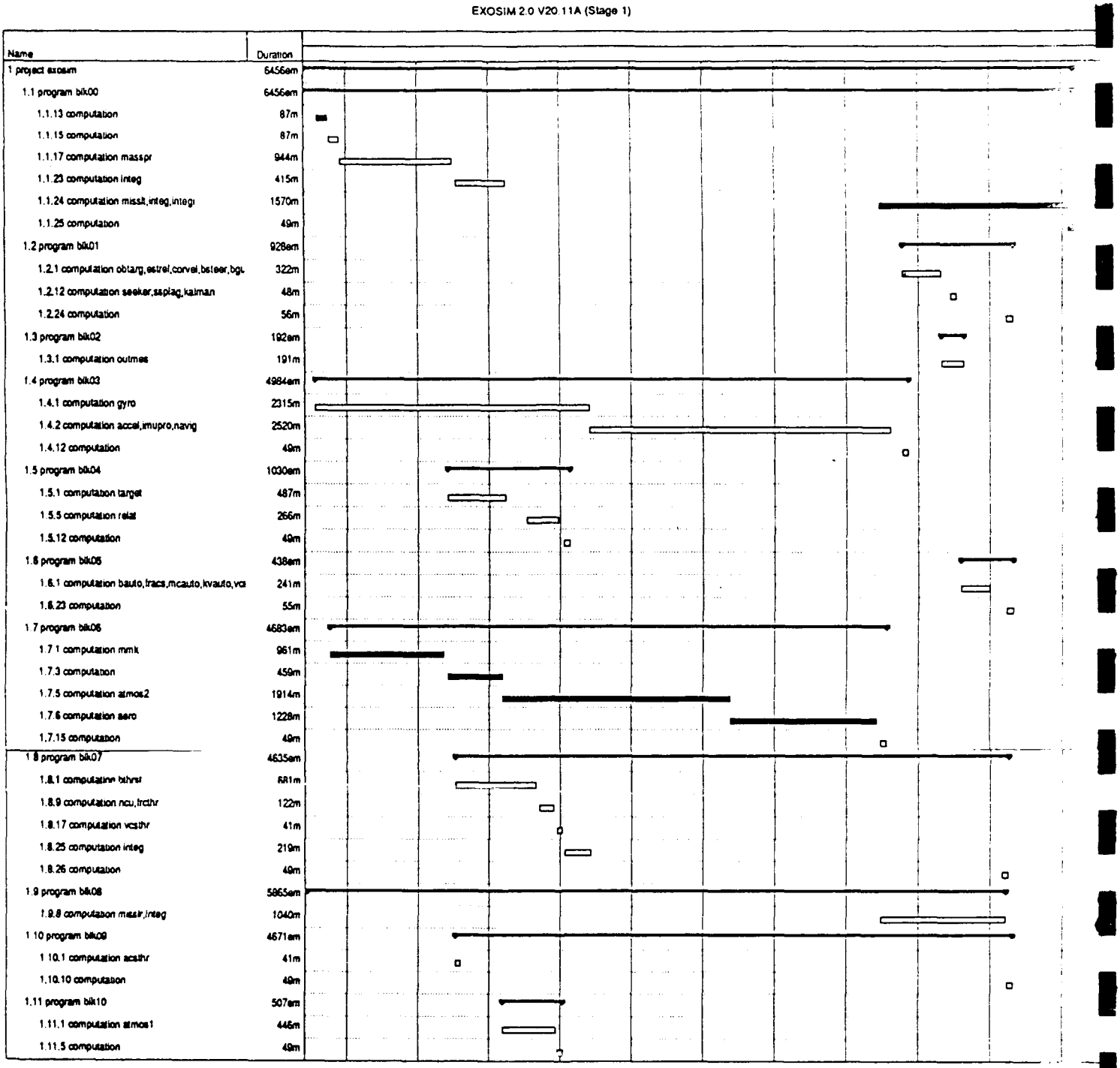

EXOSIM 2.0 V20.11A (Stage 1)
Name
Duration
1 project exosim 6456em
1.1 program blk00 6456em
1.1.13 computation 87m
1.1.15 computation 87m
1.1.17 computation massp 944m
1.1.23 computation integ 415m
1.1.24 computation missl,integ,integ 1570m
1.1.25 computation 49m
1.2 program blk01 928em
1.2.1 computation obtarg,estrel,corvel,bsteer,bgl 322m
1.2.12 computation seeker,ssplag,kalman 48m
1.2.24 computation 56m
1.3 program blk02 192em
1.3.1 computation outmes 191m
1.4 program blk03 4984em
1.4.1 computation gyro 2315m
1.4.2 computation accel,imupro,navig 2520m
1.4.12 computation 49m
1.5 program blk04 1030em
1.5.1 computation target 487m
1.5.5 computation relat 266m
1.5.12 computation 49m
1.6 program blk05 438em
1.6.1 computation bauto,tracs,mcauto,kvauto,vcl 241m
1.6.23 computation 55m
1.7 program blk06 4583em
1.7.1 computation mmk 961m
1.7.3 computation 459m
1.7.5 computation atmos2 1914m
1.7.6 computation aero 1228m
1.7.15 computation 49m
1.8 program blk07 4635em
1.8.1 computation bhrsl 681m
1.8.9 computation rcu,frcthr 122m
1.8.17 computation vcsthr 41m
1.8.25 computation integ 219m
1.8.26 computation 49m
1.9 program blk08 5865em
1.9.8 computation missir,integ 1040m
1.10 program blk09 4671em
1.10.1 computation acsthr 41m
1.10.10 computation 49m
1.11 program blk10 507em
1.11.1 computation atmos1 446m
1.11.5 computation 49m

Figure 5.12

5.1.2.9. SSV20.12

GYRO and ACCEL were split into separate partitions to make the twelve-processor version. Although these routines were not part of the critical path during stage 1 (because of the dominance of atmospheric considerations), they become problematic later in flight.

We continued to use our critical path analysis tools based on Microsoft Project. We began using idealized communication times, rather than measured times, since it should be possible to reorder the communication to achieve better-than-measured times. We delayed this manual reordering until we had a version which theoretically could run in real time. A timing chart for stage 1 (boost-phase) is shown as Figure 5.14. The parallel operation of ACCEL and GYRO is evident in the timing diagram, allowing IMUPRO and NAVIG to begin earlier.

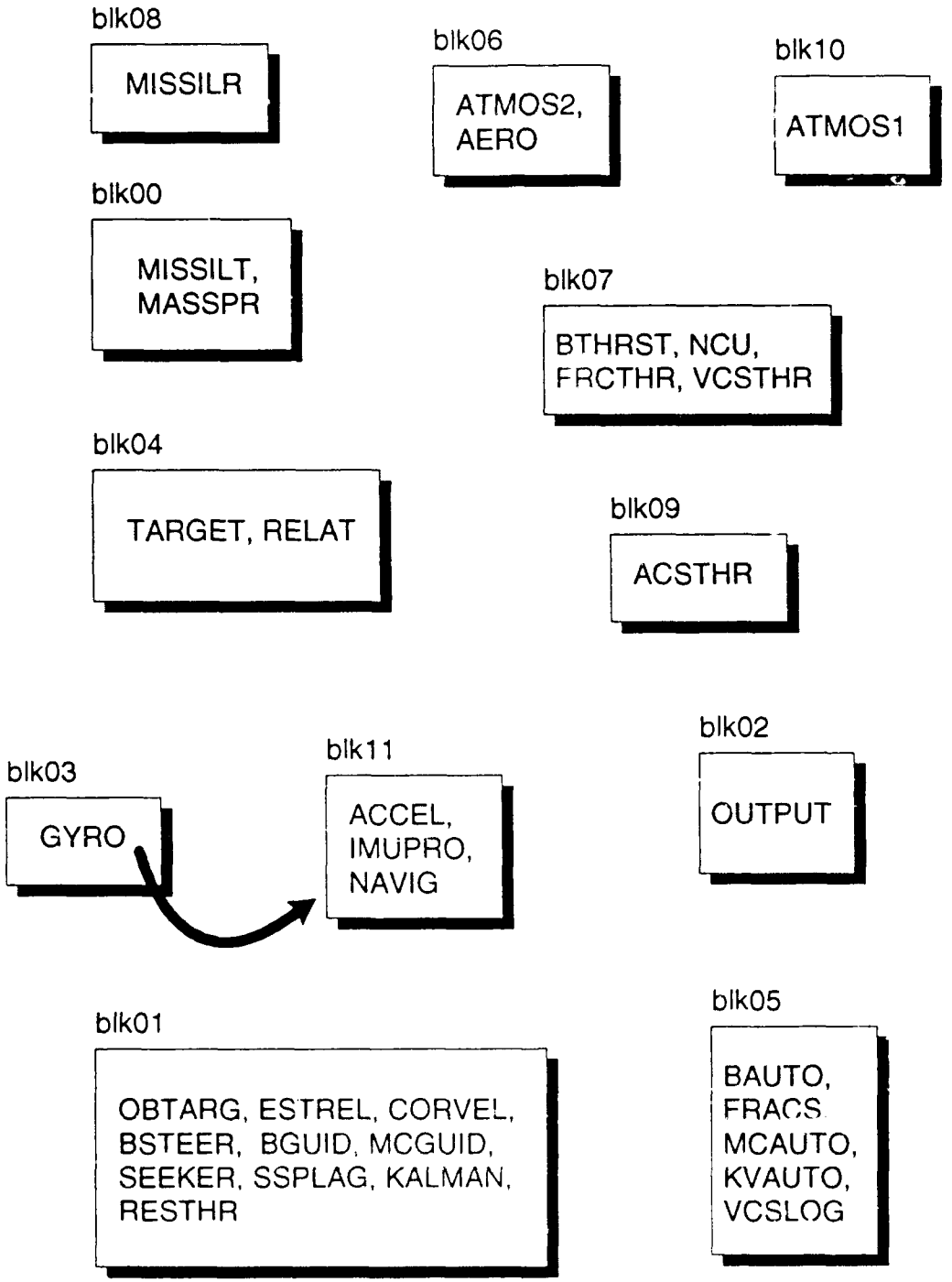


Figure 5.13: 12-partition version of EXOSIM 2.0

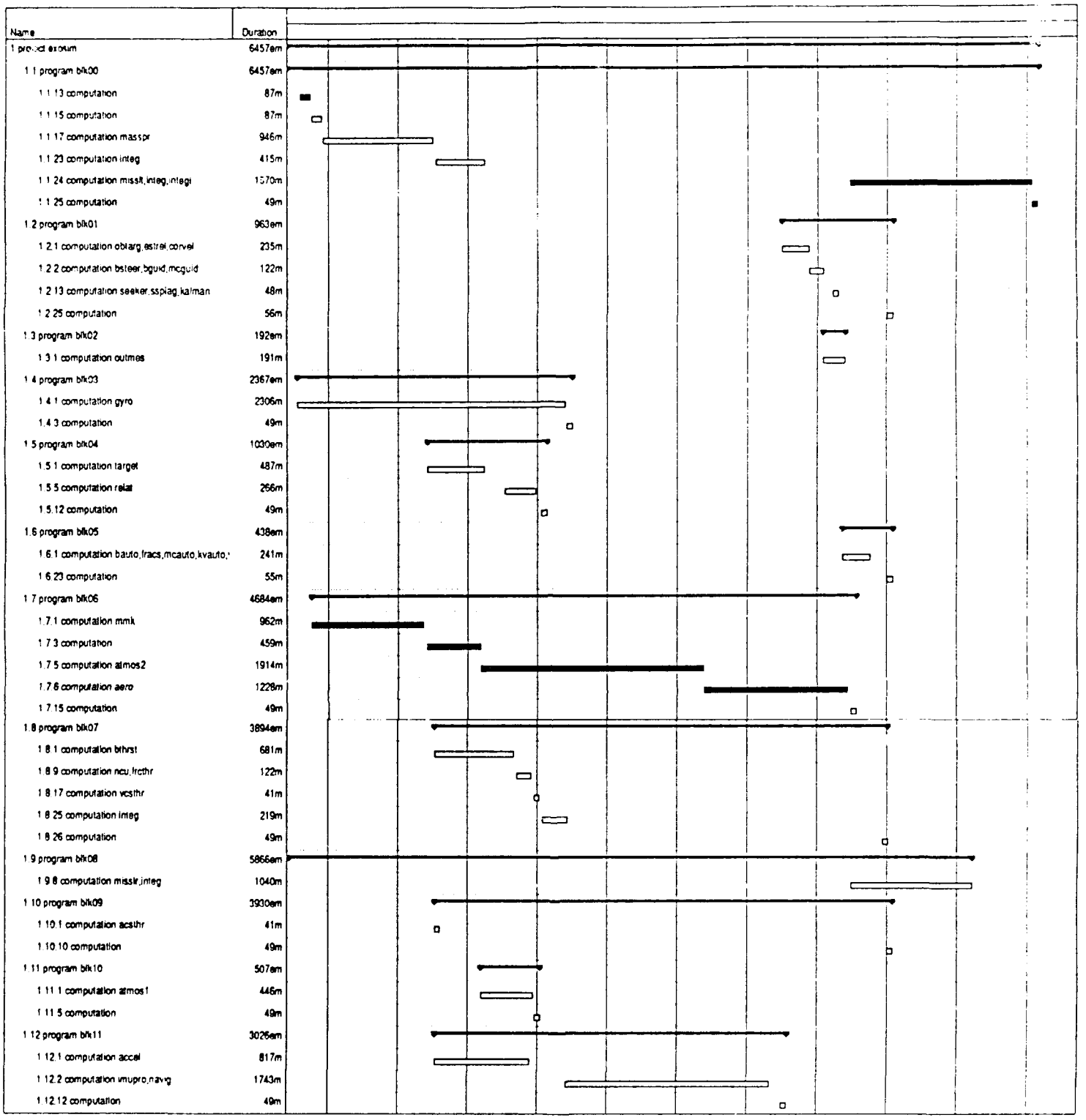

EXOSIM 2 0 V20 12A (Stage 1)
Name
Duration
1 project exosim 6457em
1 1 program blk00 6457em
1 1 13 computation 87m
1 1 15 computation 87m
1 1 17 computation masspr 946m
1 1 23 computation integ 415m
1 1 24 computation missit,integ,integi 1570m
1 1 25 computation 49m
1 2 program blk01 963em
1 2 1 computation oblarg,estrel,corvel 235m
1 2 2 computation bsteer,bguid,mcguid 122m
1 2 13 computation seeker,sspiag,kalman 48m
1 2 25 computation 56m
1 3 program blk02 192em
1 3 1 computation outmes 191m
1 4 program blk03 2367em
1 4 1 computation gyro 2306m
1 4 3 computation 49m
1 5 program blk04 1030em
1 5 1 computation target 487m
1 5 5 computation relat 266m
1 5 12 computation 49m
1 6 program blk05 438em
1 6 1 computation bauto,fracs,mcauto,kvauto,' 241m
1 6 23 computation 55m
1 7 program blk06 4684em
1 7 1 computation mmk 962m
1 7 3 computation 459m
1 7 5 computation atmos2 1914m
1 7 6 computation aero 1228m
1 7 15 computation 49m
1 8 program blk07 3894em
1 8 1 computation bthrst 681m
1 8 9 computation ncu,frcthr 122m
1 8 17 computation vcsthr 41m
1 8 25 computation integ 219m
1 8 26 computation 49m
1 9 program blk08 5866em
1 9 8 computation missir,integ 1040m
1 10 program blk09 3930em
1 10 1 computation acsthr 41m
1 10 10 computation 49m
1 11 program blk10 507em
1 11 1 computation atmos1 446m
1 11 5 computation 49m
1 12 program blk11 3026em
1 12 1 computation accel 817m
1 12 2 computation vmupro,navig 1743m
1 12 12 computation 49m

Figure 5.14

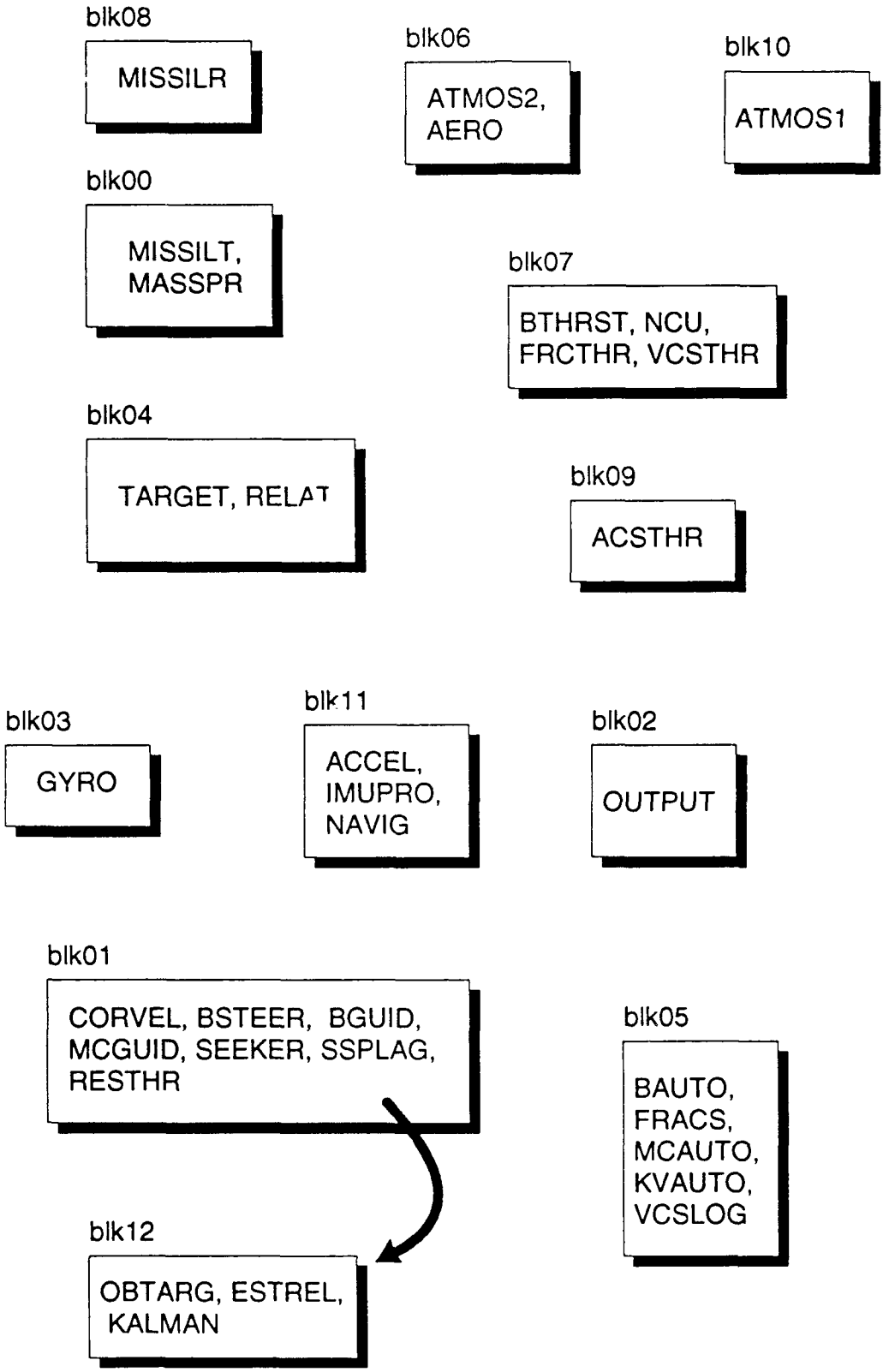


Figure 5.15: 13-partition version of EXOSIM 2.0

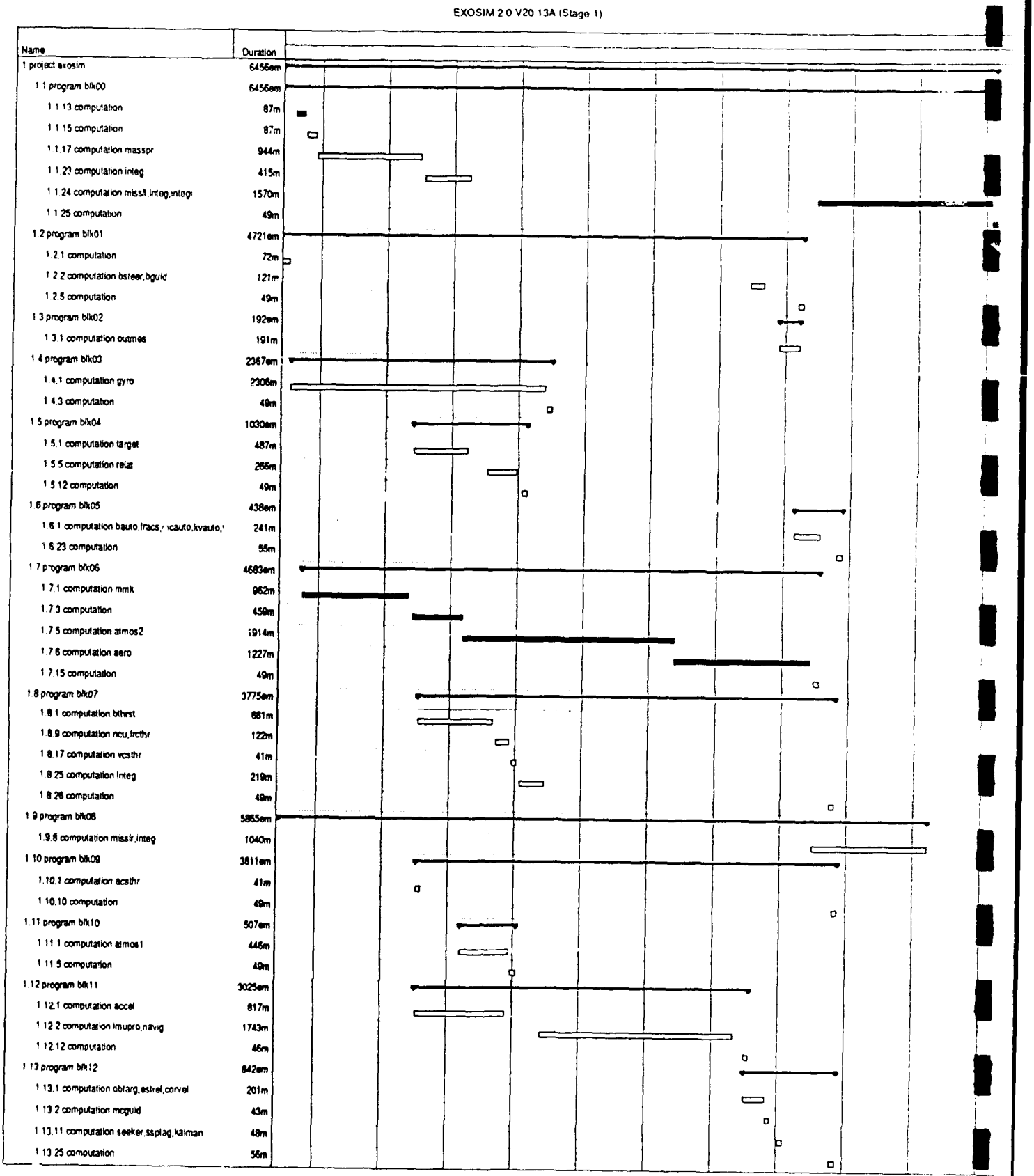

EXOSIM 2 0 V20 13A (Stage 1)
Name
Duration
1 project exosim 6456em
1 1 program blk00 6456em
1 1 13 computation 87m
1 1 15 computation 87m
1 1 17 computation massp 944m
1 1 23 computation integ 415m
1 1 24 computation missit,integ,integr 1570m
1 1 25 computation 49m
1 2 program blk01 4721em
1 2 1 computation 72m
1 2 2 computation bsteer,bguid 121m
1 2 5 computation 49m
1 3 program blk02 192em
1 3 1 computation outmes 191m
1 4 program blk03 2367em
1 4 1 computation gyro 2308m
1 4 3 computation 49m
1 5 program blk04 1030em
1 5 1 computation target 487m
1 5 5 computation relat 266m
1 5 12 computation 49m
1 6 program blk05 438em
1 6 1 computation bauto,fracs,rcauto,kvauto, 241m
1 6 23 computation 55m
1 7 program blk06 4683em
1 7 1 computation mmk 962m
1 7 3 computation 459m
1 7 5 computation atmos2 1914m
1 7 6 computation aero 1227m
1 7 15 computation 49m
1 8 program blk07 3775em
1 8 1 computation bthrst 681m
1 8 9 computation ncu,frcthr 122m
1 8 17 computation vcsthr 41m
1 8 25 computation integ 219m
1 8 26 computation 49m
1 9 program blk08 5865em
1 9 8 computation missir,integ 1040m
1 10 program blk09 3811em
1 10 1 computation acsthr 41m
1 10 10 computation 49m
1 11 program blk10 507em
1 11 1 computation atmos1 446m
1 11 5 computation 49m
1 12 program blk11 3025em
1 12 1 computation accel 817m
1 12 2 computation imupro,navig 1743m
1 12 12 computation 46m
1 13 program blk12 842em
1 13 1 computation obtarg,estrel,corvel 201m
1 13 2 computation mcguid 43m
1 13 11 computation seeker,ssplag,kalman 48m
1 13 25 computation 56m

Figure 5.16

5.1.2.11. SSV20.14

The fourteen-processor version was created by separating AERO into its own partition. As seen in Figure 5.18, this removed a substantial amount of computation from the critical path in the "blk06" partition, reducing the total simulation time by almost 20%. As noted earlier, these are still idealized times, based on actual compute times and theoretical communication times. We were able to use the project-planner timing charts, however, to create optimal orderings of communication for each stage of the flight. This information was used to generate communication priorities that were fed into the crossbar/sequencer code generation utility program. Wherever possible, the optimal ordering was made (if it did not result in an invalid ordering on any single processor). We still planned to make some changes to the individual sends and receives in a later version in order to more closely match the optimal ordering, at least for one phase of flight.

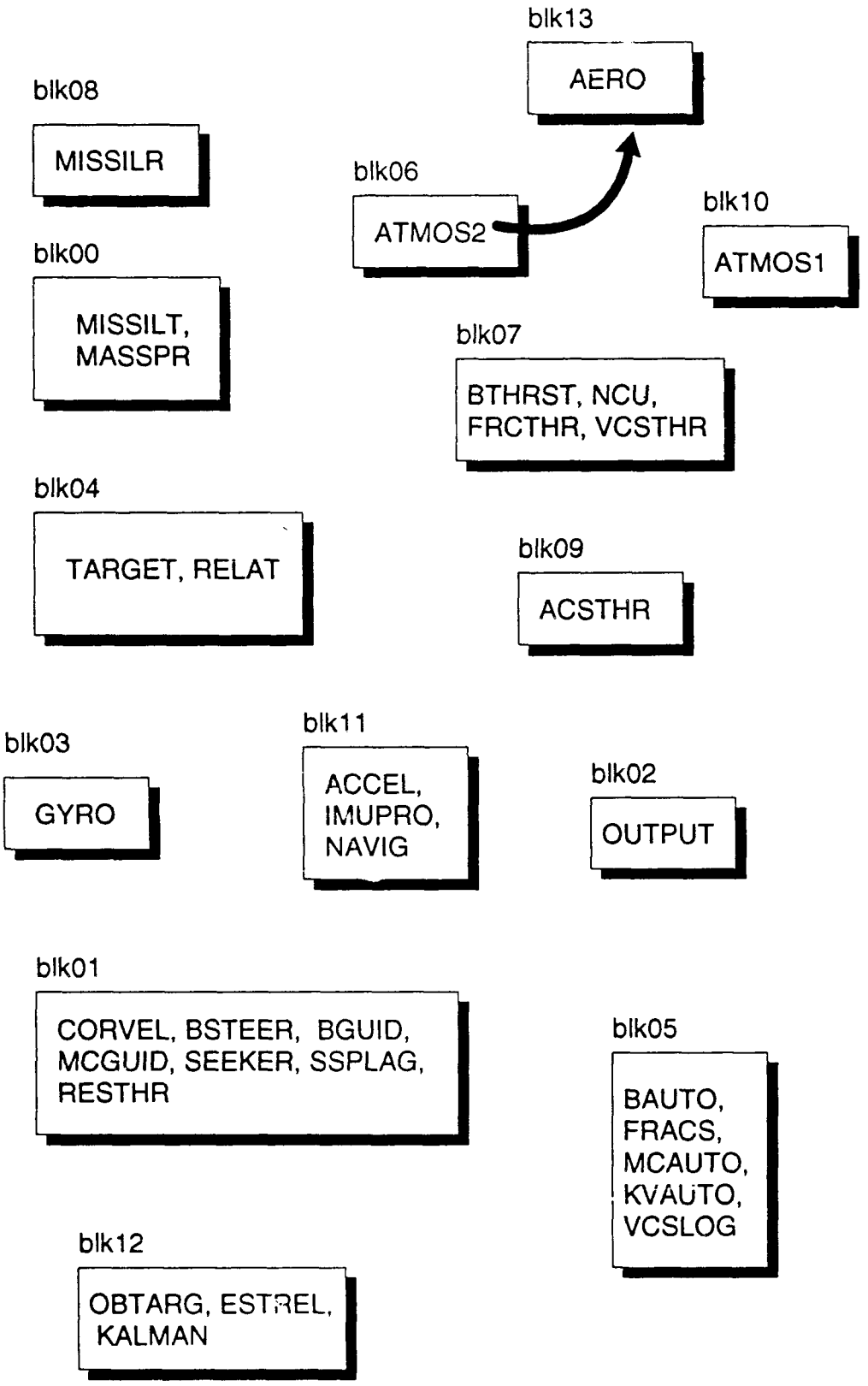


Figure 5.17: 14-partition version of EXOSIM 2.0

EXOSIM 2.0 V20.14A (Stage 1)

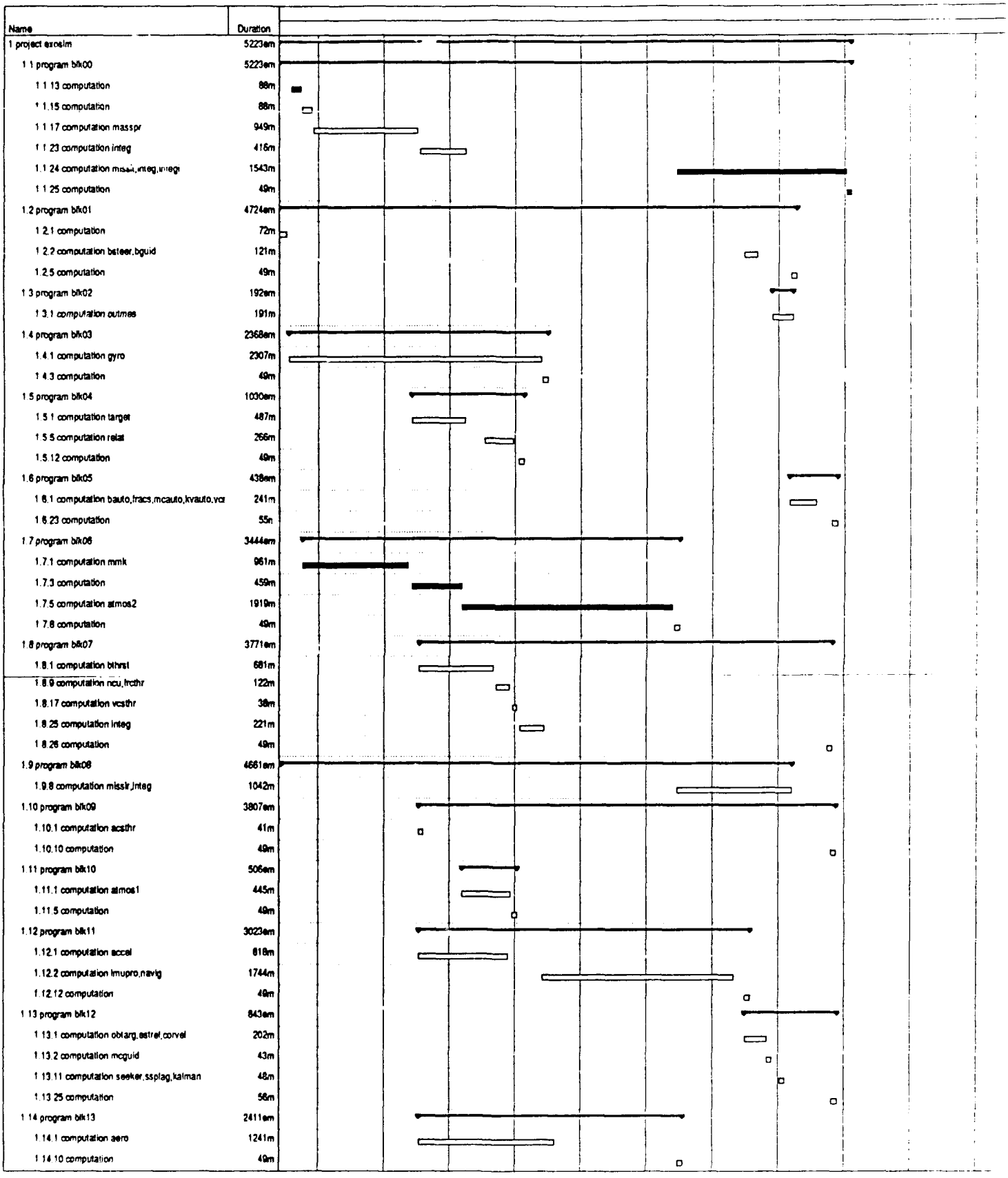


Figure 5.18

5.1.2.12. SSV20.15

The fifteen-processor version was created by splitting ACSTHR into two routines. While the previous split was made to alleviate the critical path in boost-phase (the AERO partition), this one was for the benefit of the midcourse/terminal phase (the ACSTHR partition). This sort of partitioning is typical of the problems associated with an application whose structure changes during a run — what benefits one phase of the run may not benefit another phase. In order to better illustrate the effect of this partitioning, timing charts for stage 3 (terminal phase) are provided here for both before (14-processor) and after (15-processor). (The previous 14-processor timing chart was for stage 1 only.) Figure 5.19 is the timing of the 14-processor version , and Figure 5.20 is the timing of the 15-processor version. Note that the ACSTHR computation falls out of the critical path completely as a result of this new partition. The total simulation time seems to increase, but that is only the result of using a more conservative value for each of the individual communication times.

A timing chart of stage 1 of this partitioning is given in Figure 5.23. This provides a comparison with the earlier charts, which were all for stage 1, as well as the charts which are to follow.

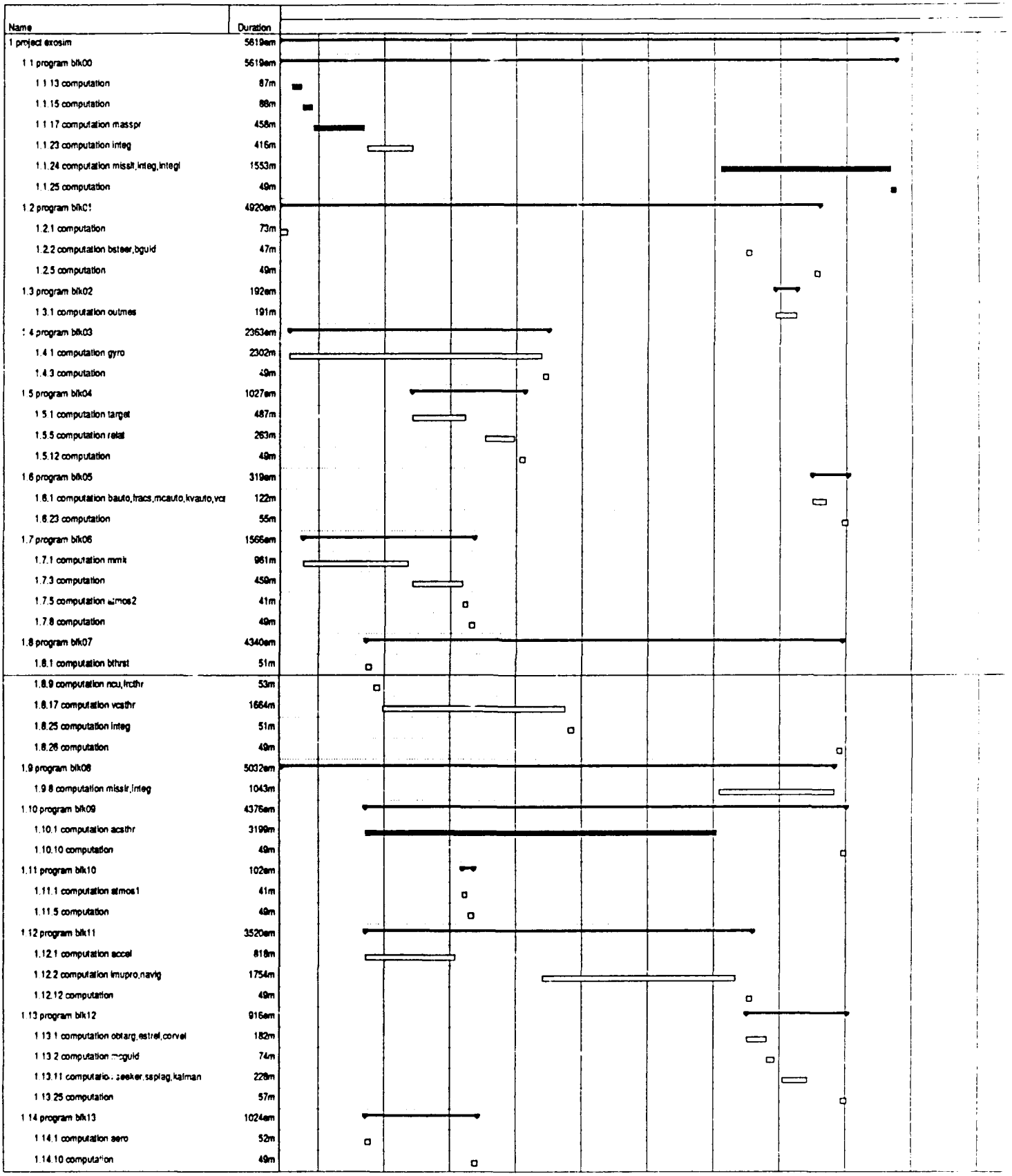

Name
Duration
1 project exosim 5619am
1 1 program blk00 5619am
1 1 13 computation 87m
1 1 15 computation 88m
1 1 17 computation masspr 458m
1 1 23 computation integ 416m
1 1 24 computation missit,integ,integl 1553m
1 1 25 computation 49m
1 2 program blk01 4920am
1 2 1 computation 73m
1 2 2 computation bsteer,bguid 47m
1 2 5 computation 49m
1 3 program blk02 192am
1 3 1 computation outmes 191m
1 4 program blk03 2363am
1 4 1 computation gyro 2302m
1 4 3 computation 49m
1 5 program blk04 1027am
1 5 1 computation target 487m
1 5 5 computation relat 263m
1 5 12 computation 49m
1 6 program blk05 319am
1 6 1 computation bauto,tracs,mcauto,kvauto,vcr 122m
1 6 23 computation 55m
1 7 program blk06 1566am
1 7 1 computation mmk 961m
1 7 3 computation 459m
1 7 5 computation atmos2 41m
1 7 8 computation 49m
1 8 program blk07 4340am
1 8 1 computation bthrst 51m
1 8 9 computation rcu,frcthr 53m
1 8 17 computation vcsthr 1664m
1 8 25 computation integ 51m
1 8 26 computation 49m
1 9 program blk08 5032am
1 9 8 computation missir,integ 1043m
1 10 program blk09 4376am
1 10 1 computation acsthr 3199m
1 10 10 computation 49m
1 11 program blk10 102am
1 11 1 computation atmos1 41m
1 11 5 computation 49m
1 12 program blk11 3520am
1 12 1 computation accel 818m
1 12 2 computation imupro,navig 1754m
1 12 12 computation 49m
1 13 program blk12 916am
1 13 1 computation obtarg,estrel,corvel 182m
1 13 2 computation mcguid 74m
1 13 11 computatio.. seeker,ssplag,kalman 228m
1 13 25 computation 57m
1 14 program blk13 1024am
1 14 1 computation aero 52m
1 14 10 computation 49m

Figure 5.19

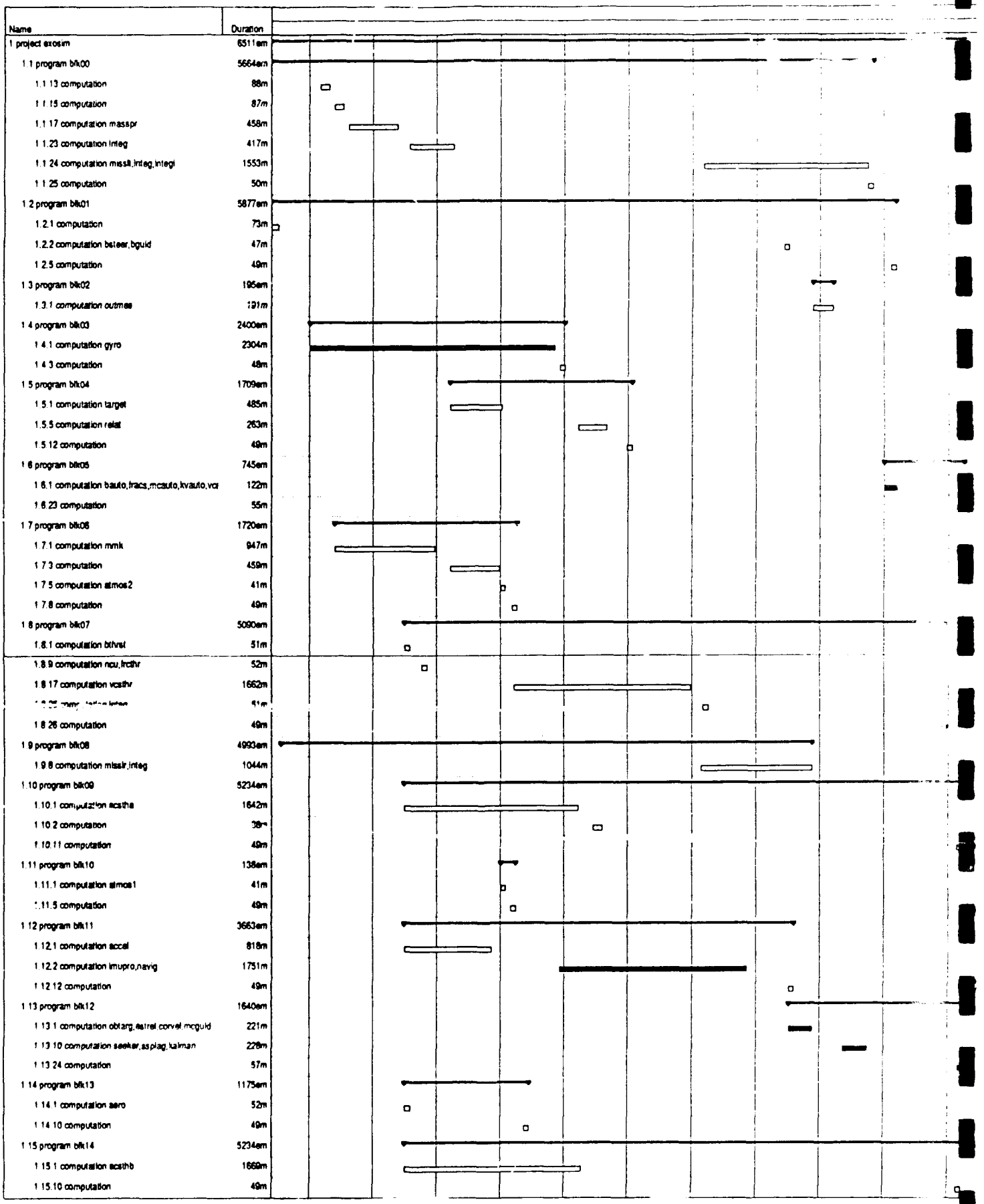

Name
Duration
1 project exosim 6511em
1.1 program blk00 5664em
1.1.13 computation 88m
1.1.15 computation 87m
1.1.17 computation massp 458m
1.1.23 computation integ 417m
1.1.24 computation mass8,integ,integl 1553m
1.1.25 computation 50m
1.2 program blk01 5877em
1.2.1 computation 73m
1.2.2 computation bsteer,bguid 47m
1.2.5 computation 49m
1.3 program blk02 195em
1.3.1 computation outmas 191m
1.4 program blk03 2400em
1.4.1 computation gyro 2304m
1.4.3 computation 48m
1.5 program blk04 1709em
1.5.1 computation target 485m
1.5.5 computation relat 263m
1.5.12 computation 49m
1.6 program blk05 745em
1.6.1 computation bauto,fracs,mcauto,kvauto,vcr 122m
1.6.23 computation 55m
1.7 program blk06 1720em
1.7.1 computation mmk 947m
1.7.3 computation 459m
1.7.5 computation atmos2 41m
1.7.8 computation 49m
1.8 program blk07 5090em
1.8.1 computation bthrst 51m
1.8.9 computation ncu,lrcthr 52m
1.8.17 computation vcsthr 1662m
1.8.26 computation 49m
1.9 program blk08 4993em
1.9.8 computation mtsslr,integ 1044m
1.10 program blk09 5234em
1.10.1 computation acstha 1642m
1.10.2 computation 39m
1.10.11 computation 49m
1.11 program blk10 138em
1.11.1 computation atmos1 41m
1.11.5 computation 49m
1.12 program blk11 3663em
1.12.1 computation accel 818m
1.12.2 computation imupro,navig 1751m
1.12.12 computation 49m
1.13 program blk12 1640em
1.13.1 computation obtarg,estrel,corvel,mcguid 221m
1.13.10 computation seeker,asplag,kalman 228m
1.13.24 computation 57m
1.14 program blk13 1175em
1.14.1 computation aero 52m
1.14.10 computation 49m
1.15 program blk14 5234em
1.15.1 computation acsthb 1669m
1.15.10 computation 49m

Figure 5.20

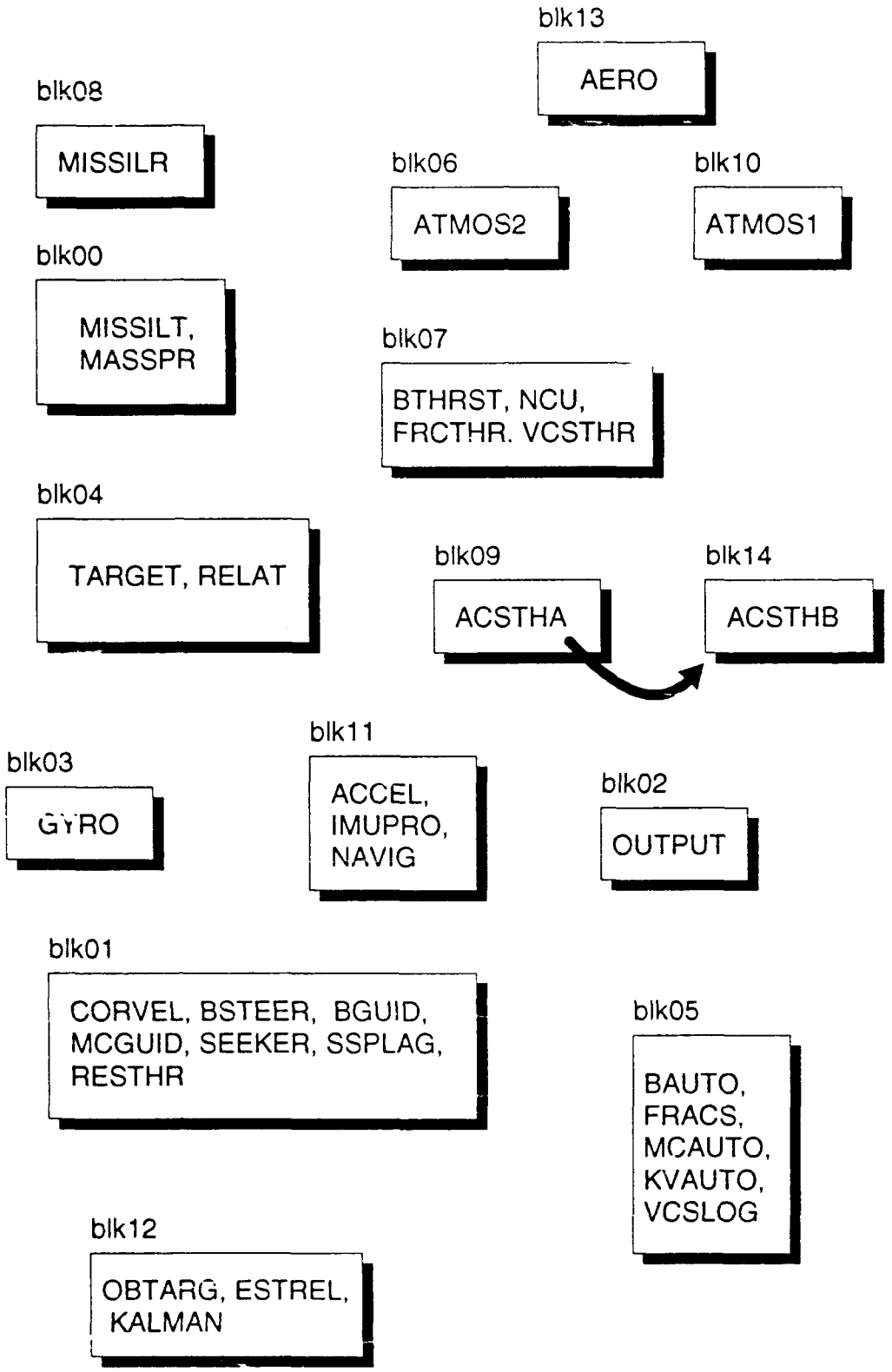


Figure 5.21: 15-partition version of EXOSIM 2.0

5.1.2.13. SSV20.16

The sixteen-processor version was created by splitting the autopilots into boost and post-boost partitions. This was done only to accomodate the FPP and produced no speed benefit. This is not especially clear from Figures 5.23 and 5.24, since we reverted to the more optimistic estimates of communication time (in anticipation of being able to convert many of the floating-point variables to shorter single-precision values). It is fairly evident, though, that the basic parallelism does not change much between these versions except near the end of the cycle, when communication tends to dominate the total execution time.

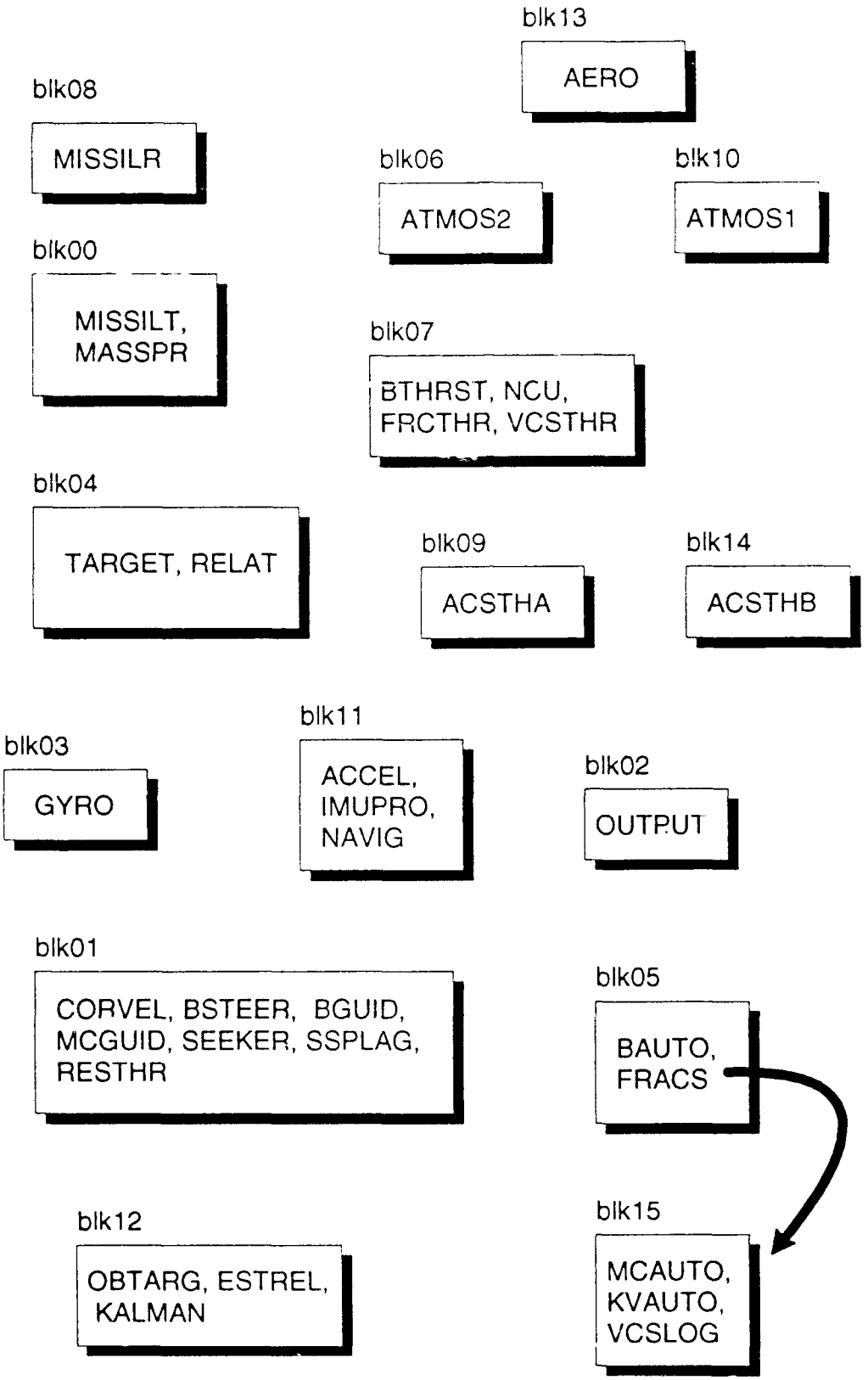


Figure 5.22: 16-partition version of EXOSIM 2.0

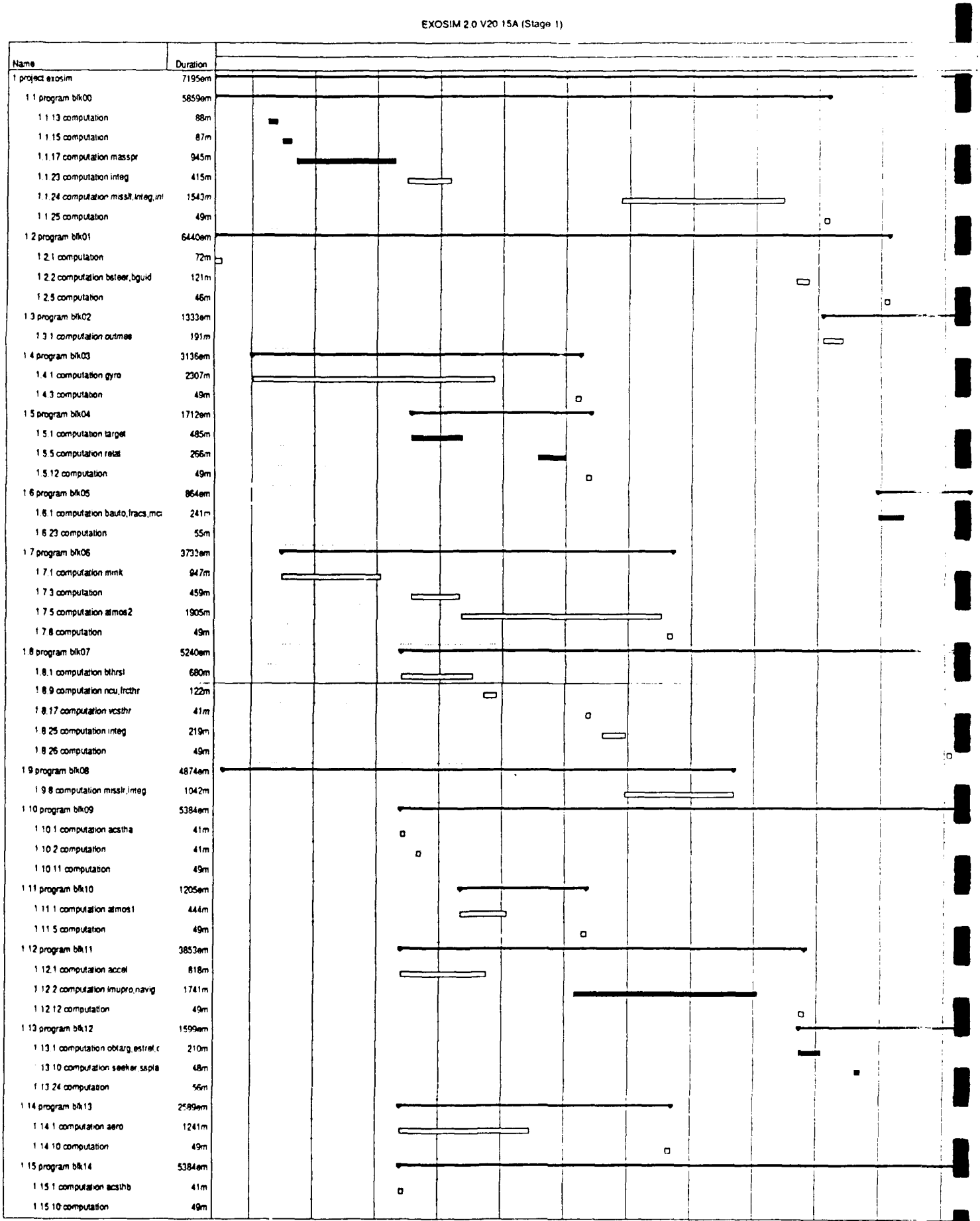

EXOSIM 2.0 V20 15A (Stage 1)
Name
Duration
1 project exosim 7195em
1 1 program blk00 5859em
1 1 13 computation 88m
1 1 15 computation 87m
1 1 17 computation masspr 945m
1 1 23 computation integ 415m
1 1 24 computation mrsslt,integ,int 1543m
1 1 25 computation 49m
1 2 program blk01 6440em
1 2 1 computation 72m
1 2 2 computation bsteer,bguid 121m
1 2 5 computation 46m
1 3 program blk02 1333em
1 3 1 computation outmes 191m
1 4 program blk03 3136em
1 4 1 computation gyro 2307m
1 4 3 computation 49m
1 5 program blk04 1712em
1 5 1 computation target 485m
1 5 5 computation relat 266m
1 5 12 computation 49m
1 6 program blk05 864em
1 6 1 computation bauto,fracs,mc 241m
1 6 23 computation 55m
1 7 program blk06 3733em
1 7 1 computation mmk 947m
1 7 3 computation 459m
1 7 5 computation atmos2 1905m
1 7 8 computation 49m
1 8 program blk07 5240em
1 8 1 computation bthrst 680m
1 8 9 computation ncu,frcthr 122m
1 8 17 computation vcsthr 41m
1 8 25 computation integ 219m
1 8 26 computation 49m
1 9 program blk08 4874em
1 9 8 computation mrsslr,integ 1042m
1 10 program blk09 5384em
1 10 1 computation acstha 41m
1 10 2 computation 41m
1 10 11 computation 49m
1 11 program blk10 1205em
1 11 1 computation atmos1 444m
1 11 5 computation 49m
1 12 program blk11 3853em
1 12 1 computation accel 818m
1 12 2 computation imupro,navig 1741m
1 12 12 computation 49m
1 13 program blk12 1599em
1 13 1 computation obtarg,estrel,c 210m
1 13 10 computation seeker,sspla 48m
1 13 24 computation 56m
1 14 program blk13 2589em
1 14 1 computation aero 1241m
1 14 10 computation 49m
1 15 program blk14 5384em
1 15 1 computation acsthb 41m
1 15 10 computation 49m

Figure 5.23

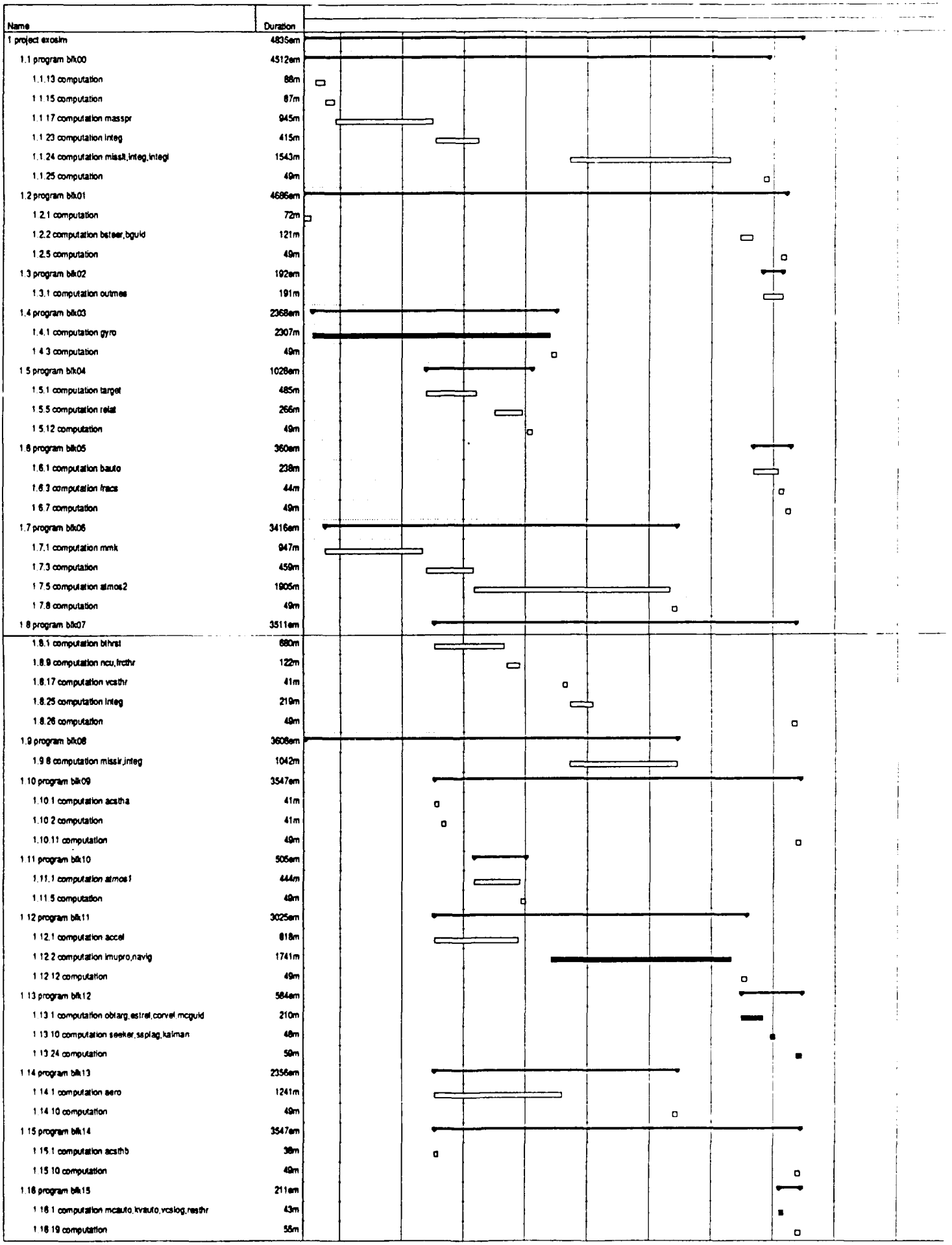

Name
Duration
1 project exosim 4835em
1.1 program blk00 4512em
1.1.13 computation 88m
1.1.15 computation 87m
1.1.17 computation masspr 945m
1.1.23 computation integ 415m
1.1.24 computation missit,integ,integi 1543m
1.1.25 computation 49m
1.2 program blk01 4686em
1.2.1 computation 72m
1.2.2 computation bsteer,bguid 121m
1.2.5 computation 49m
1.3 program blk02 192em
1.3.1 computation outmes 191m
1.4 program blk03 2368em
1.4.1 computation gyro 2307m
1.4.3 computation 49m
1.5 program blk04 1028em
1.5.1 computation target 485m
1.5.5 computation relat 266m
1.5.12 computation 49m
1.6 program blk05 360em
1.6.1 computation baute 238m
1.6.3 computation fracs 44m
1.6.7 computation 49m
1.7 program blk06 3416em
1.7.1 computation mmk 947m
1.7.3 computation 459m
1.7.5 computation atmos2 1905m
1.7.8 computation 49m
1.8 program blk07 3511em
1.8.1 computation bthrst 680m
1.8.9 computation ncu,trcthr 122m
1.8.17 computation vcsthr 41m
1.8.25 computation integ 219m
1.8.26 computation 49m
1.9 program blk08 3608em
1.9.8 computation missir,integ 1042m
1.10 program blk09 3547em
1.10.1 computation acstha 41m
1.10.2 computation 41m
1.10.11 computation 49m
1.11 program blk10 505em
1.11.1 computation atmos1 444m
1.11.5 computation 49m
1.12 program blk11 3025em
1.12.1 computation accel 618m
1.12.2 computation imupro,navig 1741m
1.12.12 computation 49m
1.13 program blk12 564em
1.13.1 computation obtarg,estref,corvel,mcguid 210m
1.13.10 computation seeker,ssplag,kalman 48m
1.13.24 computation 59m
1.14 program blk13 2356em
1.14.1 computation aero 1241m
1.14.10 computation 49m
1.15 program blk14 3547em
1.15.1 computation acsthb 38m
1.15.10 computation 49m
1.16 program blk15 211em
1.16.1 computation mcauto,kvauto,vcslog,resthr 43m
1.16.19 computation 55m

Figure 5.24

5.1.2.14. SSV21.16

We began producing another major revision, SSV21.16, based on our latest sixteen-processor version, SSV20.16a. We chose to consider this a new version (21) because we were making changes which were fundamental to the program, while not actually adding any new partitions or making changes just in a few partitions.

First, we manually split up the initialization code so that each processor initialized only what was necessary. (This refers to the explicit initialization code, not the DATA statements). After completing this task, the simulation produced the same results as before.

Then, we split up the DATA statements and all variable declarations, again so that each processor carried no more code than is necessary. The declaration statements were automatically generated by the UNIX-based utility program "DECLARE" (described below) and the data statements were automatically selected by an AWK filter.

We also created a new version that isolated the character strings and write statements scattered throughout the code to one subroutine. This change was required because the FPP/FPX C compiler does not support character strings and character I/O. During this same time, we began converting partitions to single-precision. Starting with the onboard guidance routines, two partitions were converted successfully. One partition contained the boost guidance and steering routines and the other contained the boost autopilot.

The general procedure for converting to single precision was determined. We began to convert partitions one at a time, isolating the single-precision modifications to only the partition under consideration. Previously, it was necessary to simultaneously change several partitions in order to match variable usage. The new method proceeded in two major stages on each partition. In the first step, we simply linked a new set of communication routines which just truncate double-precision values as they cross the partition boundary. No editing or recompilation is necessary for the affected processor, since it still sees the truncated values as double precision. This step simply verifies that no information going in or out of the partition really needs to be double precision -- it does not show the effect of lowered precision on the partition's cumulative calculations.

In the second step, the processor's code is modified as required to actually perform only single-precision calculations. At the same time, another new set of communication routines is linked to automatically pad the single-precision values into double-precision values which are presented to the rest of the simulation. This step verifies that the partition will truly run at the lower precision, but it delays the need for editing other partitions to accept the single-precision format. These two steps were repeated for each partition which we decided to convert.

A minor bug had crept into our 3- through 16-processor EXOSIM 2.0 simulations, causing the sequencing of burns to be different from the single-processor version. We reran the UNIX based

USAGE, COMBINE, and SUMMARY programs on V19.3c and found that the SUMMARY.TXT file contained 10 variables that were being assigned by more than 1 processor concurrently. We had looked at each of these variable conflicts during the development of V19.3c and had determined that they would not be a problem.

We reexamined each one of these conflicts to determine if our original analysis was correct. After some checking, the variables FLTC, FLTCP, & FLTCY (in VCSTHR) were found to be the problem. We split VCSTHR into VCSTH1 & VCSTH2 and moved the VCSTH2 part to the correct processor. The corrected V19.3c and V21.16c simulations were tested and produced the expected output. It should be noted that the automated analysis tools had correctly warned us of the FLTC, FLTCP, & FLTCY variable conflicts but human analysis had failed to correctly determine whether the conflicts were really a problem.

We moved the unclassified portions of the V21.16c FORTRAN source to the SUN host, where they were converted and compiled. This pointed out, much as expected, that there were three partitions would have to be split up further in order to run on the FPPs. There were also three other partitions which were slightly too large, but can probably be trimmed down without additional partitioning. At least one of the three large partitions will probably never be running on an FPP, anyway.

We also started working on the capability of running FPP/FPX code (generated for the FPP/FPX PFP on the SUN Unix system) on the 286/386 PFP. The ability to download and start FPP/FPX code was there, but for some unknown reason, only simple programs worked. We wanted to get this working so that we could test one FPP/FPX program block at a time while running the remaining program blocks unmodified on 286s or 386s.

5.1.2.15. SSV22.16

We then began to generate version 22 (specifically, SSV22.16a), which was to eliminate most of the double-precision calculations. We repeated the conversions of the boost-stage guidance and control partitions, using a previous conversion as a model. With each partition that we converted, we were able to isolate the only variables which had to be passed in as double precision, using the methodology that has been outlined before.

In summary, the following partitions were initially converted to single precision:

Boost guidance
Boost autopilot
Output
Gyro
Atmosphere 1
Atmosphere 2
Boost and VCS thrusters
ACS thrusters 1
ACS thrusters 2
Missile rotational states
Aerodynamics

Each of these has been tested, and the simulation behaves normally. Because of the method that was used, we could easily switch back and forth between single and double precision for any given partition with no recompilation. It is thus possible to test different combinations, in the event that there turns out to be some cumulative effect as more partitions are converted.

The following partitions had double-precision requirements that could not be eliminated at the time:

Midcourse/terminal guidance
Midcourse/terminal autopilot
Accelerometers/Navigation
Missile translational states
Target and relative states

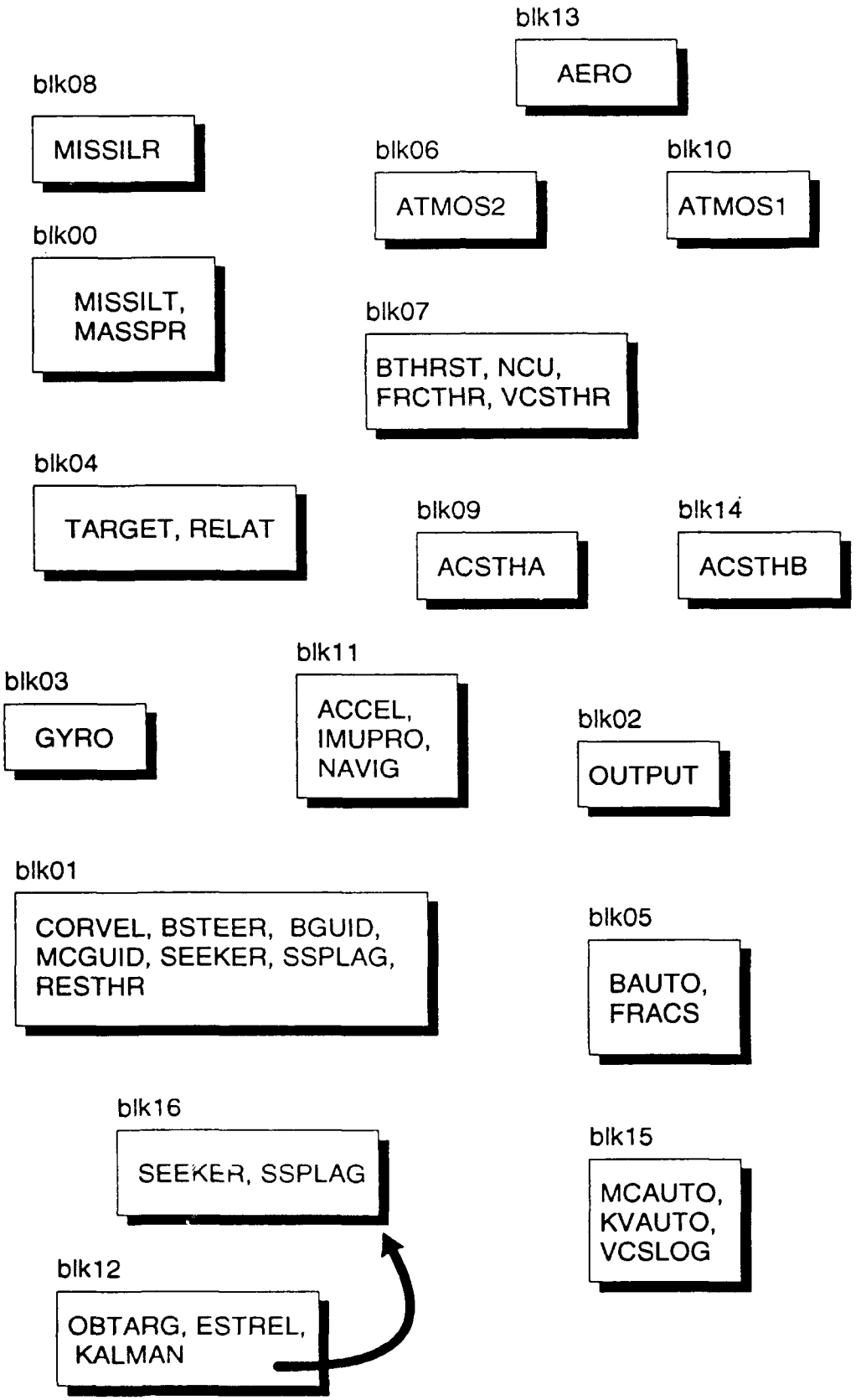


Figure 5.25: 17-partition version of EXOSIM 2.0

1. SSV22.17 and SSV22.18

We completed a new 17-processor version, SSV22.17a, by splitting out SEEKER and SSPLAC from the target and relative states partition. This would eventually have had to be done in order to interface properly with an external seeker, and it also reduced the size of one of the partitions which was too large for an FPP. We analyzed the communication and got this version running correctly.

We then converted the new partition to single precision, making version SSV22.17b. This has been verified to produce the correct output. The larger partition that remained was still one of the few remaining double precision ones, so we split out all of the double precision requirements to another partition, creating the 18-processor version SSV22.18a. This was tested and verified.

The single-precision conversion was made on one of the resulting partitions, making version SSV22.18b.

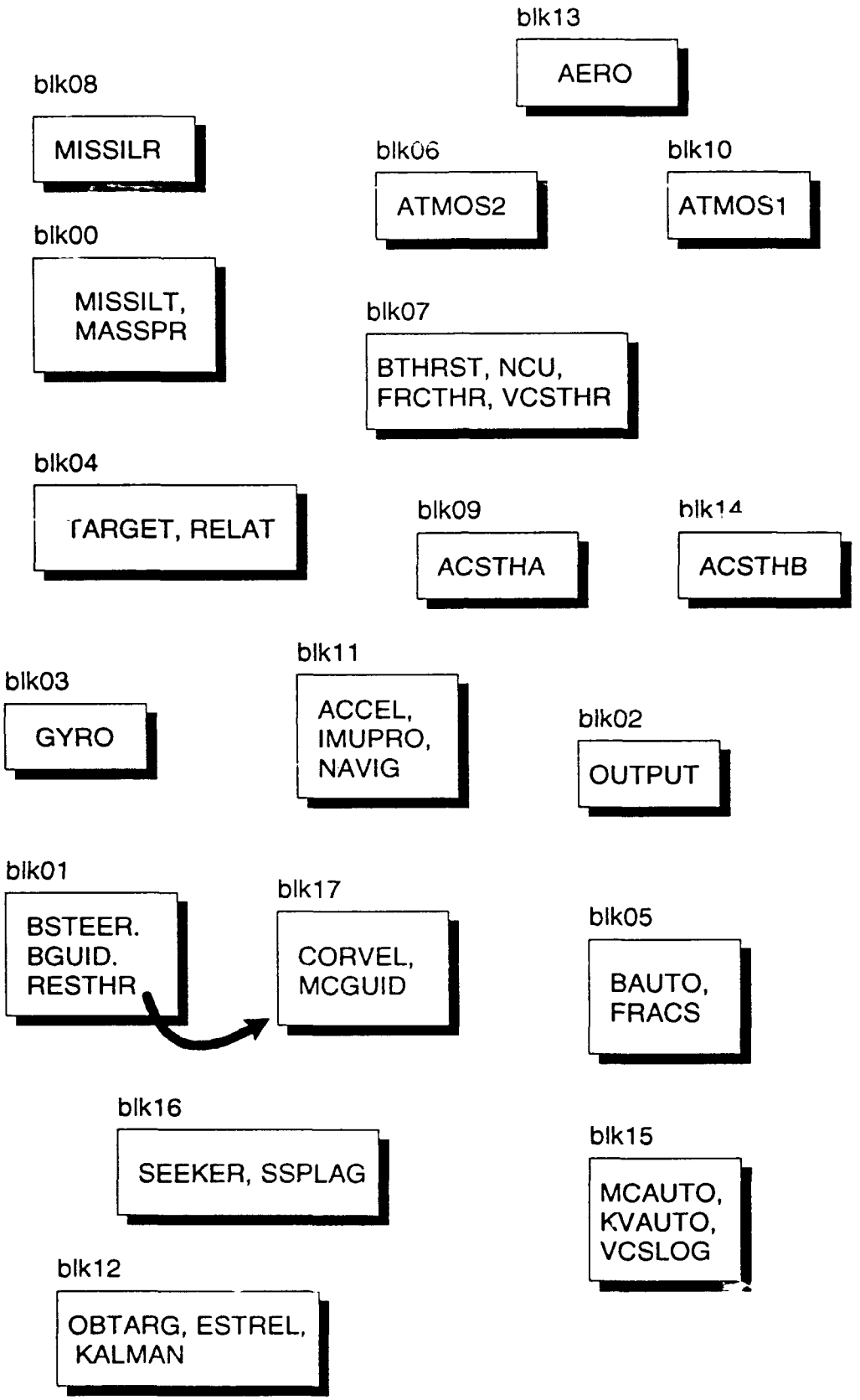


Figure 5.26: 18-partition version of EXOSIM 2.0

5.1.2.16. SSV22.19

This left only two partitions which were too big for an FPP. One of these is currently targeted for a 386 processor, but the other was fairly easy to split. This was done, resulting in the 19-processor version SSV22.19a, which produced identical output.

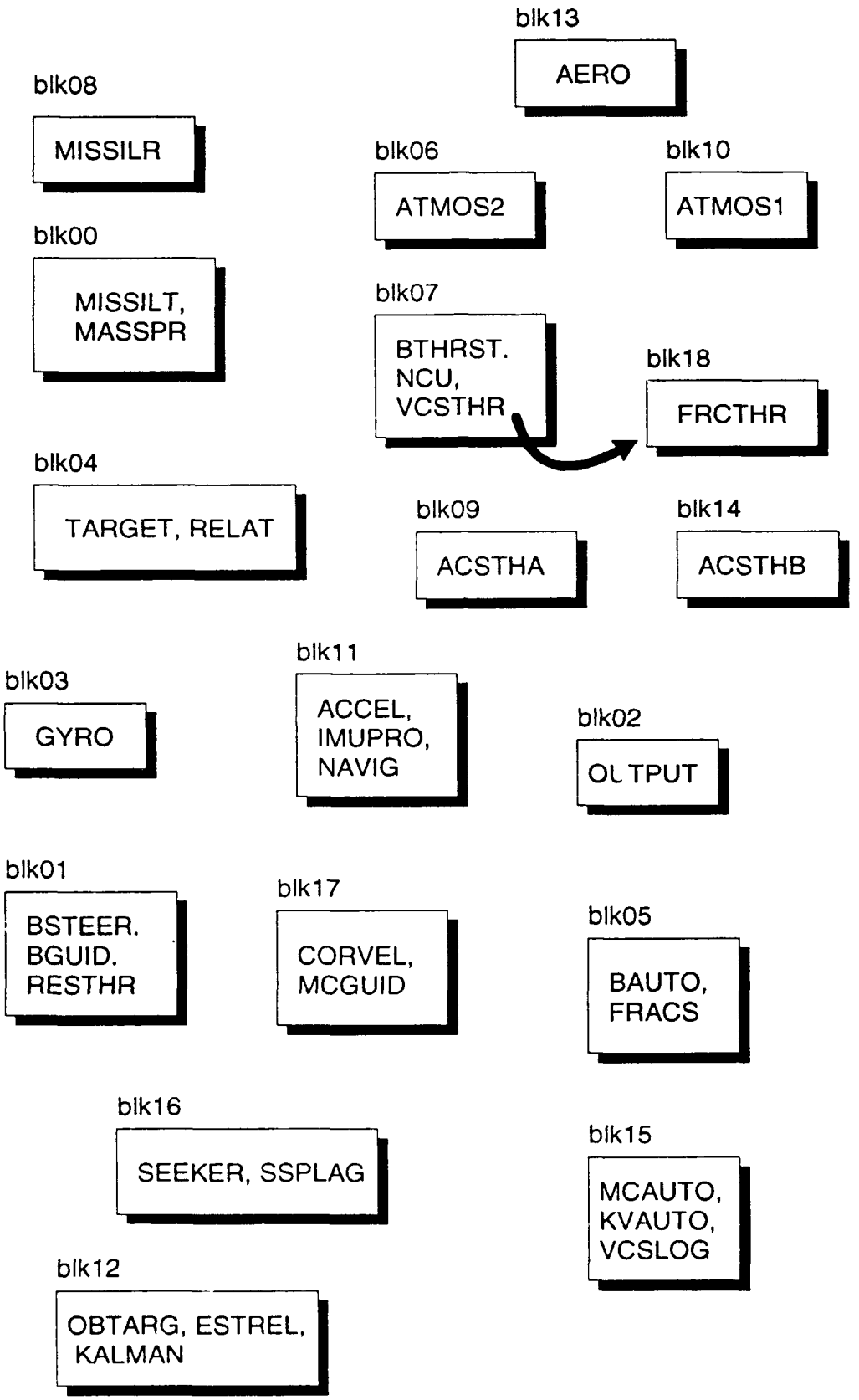


Figure 5.27: 19-partition version of EXOSIM 2.0

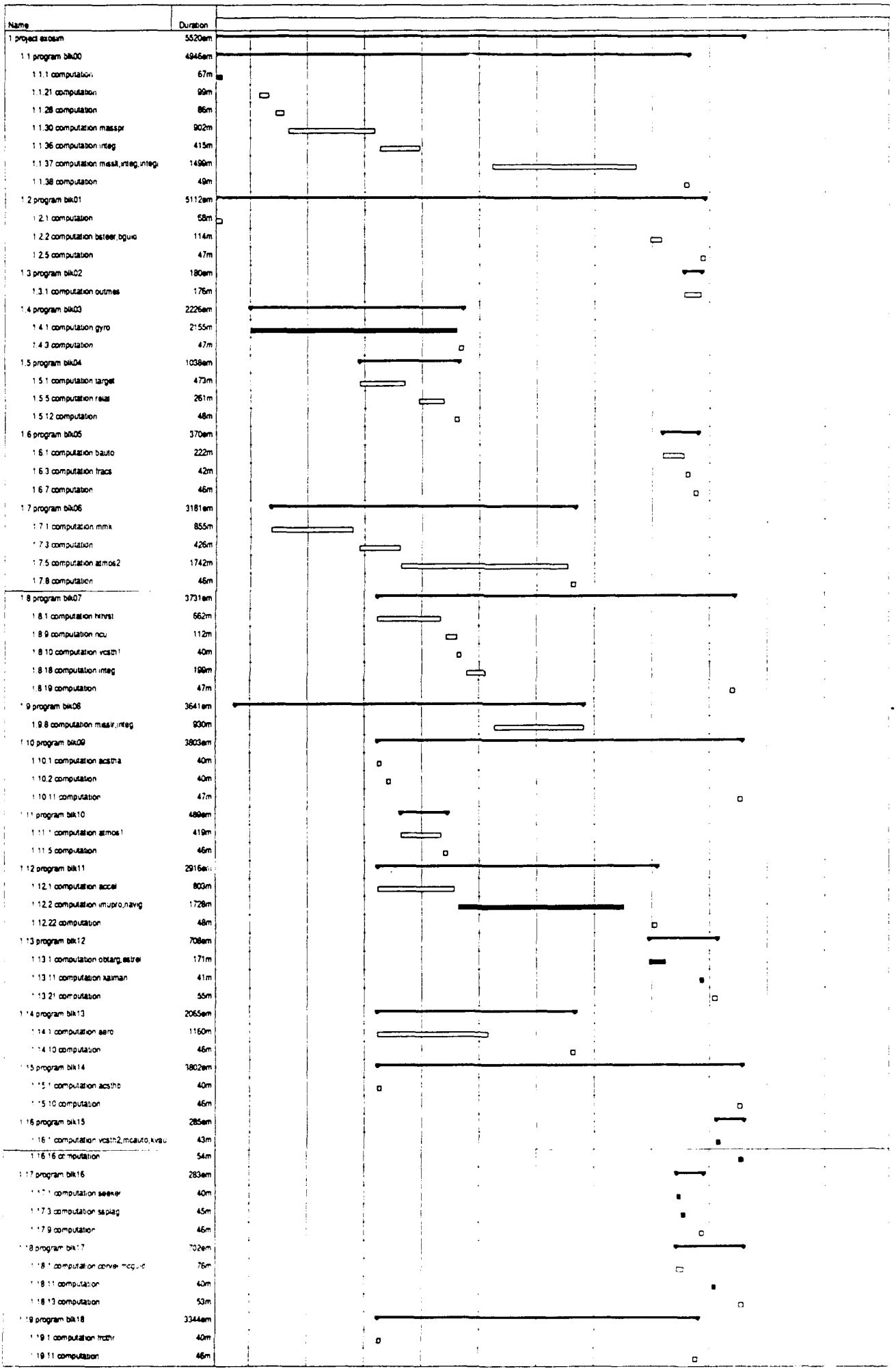

Name
Duration
1 project exosim 5520em
1.1 program blk00 4946em
1.1.1 computation 67m
1.1.21 computation 99m
1.1.28 computation 86m
1.1.30 computation massprr 902m
1.1.36 computation integ 415m
1.1.37 computation massit,integ,integi 1499m
1.1.38 computation 49m
1.2 program blk01 5112em
1.2.1 computation 58m
1.2.2 computation bsteer,bguid 114m
1.2.5 computation 47m
1.3 program blk02 180em
1.3.1 computation outmes 176m
1.4 program blk03 2226em
1.4.1 computation gyro 2155m
1.4.3 computation 47m
1.5 program blk04 1038em
1.5.1 computation target 473m
1.5.5 computation relat 261m
1.5.12 computation 48m
1.6 program blk05 370em
1.6.1 computation bauto 222m
1.6.3 computation fracs 42m
1.6.7 computation 46m
1.7 program blk06 3181em
1.7.1 computation mmix 855m
1.7.3 computation 426m
1.7.5 computation atmos2 1742m
1.7.8 computation 46m
1.8 program blk07 3731em
1.8.1 computation hmrst 562m
1.8.9 computation ncu 112m
1.8.10 computation vcsth1 40m
1.8.18 computation integ 199m
1.8.19 computation 47m
1.9 program blk08 3641em
1.9.8 computation massir,integ 930m
1.10 program blk09 3803em
1.10.1 computation acstha 40m
1.10.2 computation 40m
1.10.11 computation 47m
1.11 program blk10 489em
1.11.1 computation atmos1 419m
1.11.5 computation 46m
1.12 program blk11 2916em
1.12.1 computation accel 803m
1.12.2 computation imupro,navig 1728m
1.12.22 computation 48m
1.13 program blk12 708em
1.13.1 computation obtarg,estrel 171m
1.13.11 computation kalman 41m
1.13.21 computation 55m
1.14 program blk13 2065em
1.14.1 computation aero 1160m
1.14.10 computation 46m
1.15 program blk14 1802em
1.15.1 computation acsthb 40m
1.15.10 computation 46m
1.16 program blk15 285em
1.16.1 computation vcsth2,mcauto,kvau 43m
1.16.16 computation 54m
1.17 program blk16 283em
1.17.1 computation seeker 40m
1.17.3 computation saplag 45m
1.17.9 computation 46m
1.18 program blk17 702em
1.18.1 computation cprver mcgain 76m
1.18.11 computation 40m
1.18.13 computation 53m
1.19 program blk18 3344em
1.19.1 computation frcthr 40m
1.19.11 computation 46m

Figure 5.28

At the same time, we reexamined the continuing problem of uninitialized variables in the original version of EXOSIM 2.0. This reemerged as a problem when we attempted to compile the code on the Transputer system (as a cross-check). A reexamination of the 1-processor and 16-processor versions of EXOSIM V2.0 with the "new" version of INITIAL (described elsewhere) spotted numerous uninitialized variables. The following code fragment is a simplified version of the actual code, because all of these assignments were hidden inside subroutines.

```
IF ( T .GE. 5 )
        FRCX = ...
        FRCY = ...
        FRCZ = ...
END IF
IF ( T .GE. 10 )
        FXACS = ...
        FYACS = ...
        FZACS = ...
        FXVCS = ...
        FYVCS = ...
        FZVCS = ...
END IF
        FX = FRCX + FXACS + FXVCS
        FY = FRCY + FYACS + FYVCS
        FZ = FRCZ + FZACS + FZVCS
```

As this shows, the FRC_ variables are uninitialized until T >= 5. Also, the F_ACS and F_VCS variables are also uninitialized until T >= 10. Consequently, the values of FX, FY, FZ are suspect since they are computed sometimes with unknown values.

Unfortunately, the original EXOSIM V2.0 from Coleman Research contained many examples of this type of programming. It is essential that this be fixed because it can cause some of the hardest bugs to track down. When one considers how we move programs from 286/386 to FPP/FPX processors, we don't want to introduce any unknowns into the problem by leaving variables uninitialized.

The UNIX-based DECLARE program (described elsewhere) was modified to fix this problem by assuring that every variable used in the program has an initial value. We used DECLARE in order to generate a new 1-processor version (SSV18.1c) where all of the previously uninitialized variables (1,167 to be exact) are initialized to zero with DATA statements.

The new version was then tested on the DEC ULTRIX system with both the VAX FORTRAN Compiler, ULTRIX FORTRAN Compiler, the INTEL iRMXII system with the FORTRAN 286 Compiler, and the IBM PC with the Transputer FORTRAN Compiler. In all cases, the new version worked perfectly.

The real test came when a 386 processor's memory was filled with NOT-A-NUMBER (NAN) and then loaded and executed the program. The old version crashed immediately because the 387 tried to do an operation with NAN. The new version ran to completion without a problem.

We also began to support the integration of the Seeker Scene Emulator. Initially, this was be done with the post-boost-only version of EXOSIM being written by Richard Pitts and Philip Bingham. We worked with Andy Henshaw, implementing Richard's seeker partition on the transputers. Steve provided a syntax translation of the EXOSIM V2.0 seeker routines, using his FORTRAN-TO-OCCAM translator. Andy completed the conversion to Occam and debugged

the code. We then tested our connection to the FPP-based PFP and were able to run the seeker partition on an external transputer. The code seemed to run correctly, but not identically to the FPP version. Presumably, this was due to differences in the floating-point precision in the random number generator and elsewhere.

At this point, it became necessary to trim out all of the extra communication cycles so that we could get accurate timing and begin converting blocks to the FPPs. We created a new version, SSV22.19b, in which we manually resolved most of the precision differences between blocks. Prior to this, all variables were being sent as double precision, even when they were calculated as single precision, so conflicts never occurred. We had to introduce a few new communication routines to make precision conversions, and we modified the NETWORK program to correctly handle these new routines. A new crossbar program was generated, and the version produced the correct output.

We then began to make final preparations for moving the code to FPPs. We trimmed out all of the unnecessary receiving of single-precision values as doubles, and we eliminated all instances of variables being sent as doubles and used at different precisions on different processors. This latter step involved quite a bit more double-to-single conversion on the five remaining double-precision partitions, eventually producing version SSV22.19f.

We continued to work with version SSV22.19f, which had all of the necessary precision conversion completed. In preparation for porting individual partitions to the FPP, We prepared a timing version of the code, something we have not done since before the precision conversion began. It was not possible to extrapolate the 386 timings exactly to FPP timings, but our best estimate was that we would be close to real time, but not quite there.

Based on the timing charts generated by Microsoft Project, we then reordered much of the communication to balance the performance across stages of flight. We have tested this version and retimed it, replacing the previous SSV22.19f. A timing chart for stage 1 (boost-phase) of SSV22.19a is shown as Figure 5.28. Once again, we adjusted the theoretical communication times, taking into account the mixture of single- and double-precision variables. This resulted in an apparent lengthening of the total time relative to the last timed version, SSV20.16a, but the newer version actually ran much faster (within about a factor of four or five of real time, which was very good for the 386-based processors).

We have, at least for the time being, leveled out at 19-processor implementations of EXOSIM 2.0. We created versions SSV22.19g and SSV22.19h. Version 22.19g was generated by the UNIX-based DECLARE program. We compiled and tested the new version with main programs that now include only the variables required with each variable assigned an initial value with a data statement. Version 22.19h fixed a few of the partitions that would not otherwise fit on the FPP or FPX boards and is described in more detail below.

As noted in the past, additional partitions might have to be made simply to make EXOSIM fit on the FPP and FPX processors. Version SSV22.19g was ported to the Sun-hosted PFP, and all of the blocks were translated to C (using F2C), then compiled and linked (using the new VICLD program) to determine their memory requirements. The following table summarizes the results:

PROGRAM / TYPE	CODE	DATA	PROBLEM
SSBLK00 FPX	57708	4700	<-- code size
SSBLK01 FPP	37032	1496	
SSBLK02 FPP	5892	3356	
SSBLK03 FPP	28224	2040	
SSBLK04 FPX	56124	8140	<-- code size
SSBLK05 FPP	30696	10356	<-- data size
SSBLK06 FPP	21804	2212	
SSBLK07 FPP	30132	3228	
SSBLK08 FPP	28560	2060	
SSBLK09 FPP	28656	1948	
SSBLK10 FPP	7632	1152	
SSBLK11 FPX	78516	4880	<-- code size
SSBLK12 FPX	62964	4232	<-- code size
SSBLK13 FPP	14820	8532	<-- data size
SSBLK14 FPP	28116	1892	
SSBLK15 FPX	135852	4292	<-- code size
SSBLK16 FPP	22872	2352	
SSBLK17 FPP	34668	2200	
SSBLK18 FPP	50388	2144	<-- code size

Some of these problems were eliminated in the next version, SSV22.19h. The data sizes of both SSBLK05 and SSBLK13 were reduced to acceptable levels by eliminating some needlessly duplicated arrays (which were in the original single-processor version). The code size of

SSBLK18 was reduced by making some array parameters local to the routine FRCTHR, since the calling program did not reference them.

The remaining problems are more difficult, requiring either more partitioning or improved compilers that generate more efficient (and perhaps faster) code. We are hesitant to make partitions which would otherwise be unnecessary and contribute to additional communication overhead.

As a separate effort, Richard Pitts and Philip Bingham have begun to use the most recent complete version of EXOSIM developed by Steve Wachtel and Tom Collins as the basis of a boost-phase-only version of EXOSIM 2.0. This requires mainly deleting some of the code that is required only in the later stages of flight, along with some associated communication. This also alleviates much of the memory problems described earlier.

5.2. LEAP

Beginning in January 1991, we began working with Brian Stevens at GTRI in Cobb County. We discussed the LEAP program and GTRI's approach to its simulation. Hughes has both an emulator (running actual LEAP code) and a simulation (used internally only). Brian attempted to stay in contact with Hughes and make changes to his simulation accordingly. (There was no mechanism by which design changes are automatically routed to GTRI.)

An earlier version of GTRI-LEAP was rather large and unwieldy, probably much like EXOSIM, but Brian has concentrated on making a smaller, faster program. Much of the programming had been done by an assistant, and Brian was not yet completely confident that it was correct. He was also still making some organizational changes in the code, but had other responsibilities that prevented him from spending much time on it.

Brian has developed his own FORTRAN executive (main) program for calling the LEAP subroutines. This program is designed to be generic enough to be useful for other simulations of continuous/discrete hybrid systems, but it is not particularly suited to our needs. For this reason, we will have to write our own main program, some general math routines, and integration routines. There are two LEAP subroutines. One of them, "F," includes all of the continuous time state variables and all of the corresponding derivative calculations. The state variables are grouped as a single vector, which helps to identify them. The other function, "D," performs the discrete time calculations.

The fastest sampling rate in the actual LEAP vehicle is 3600 Hz, corresponding to the accelerometers and gyros. This has been chosen as the integration time step, and all discrete events (seeker data, PWM, quaternion and control calculations) occur at even multiples of this rate (either 60 Hz or 360 Hz). The F and D subroutines are both designed to be called at every time step, since the discrete events are scheduled within the D routine by a time-step counter (both from a standpoint of frequency and phase).

The concise state-variable representation is suitable for the PFP, and the two-subroutine implementation gives us a good start at partitioning into at least two processors. The complete implementation of a parallel version, however, will take some time, though probably not nearly as long as EXOSIM. We have already written the main program, and he has modified the integration and 3-by-3 matrix multiplication routines from EXOSIM to be usable. We also verified that the gravity calculations were correct, since the gravitational contribution had been disabled in the GTRI version of LEAP. We currently have only the "F" routine, which did compile and run, but not all variables were initialized correctly. All of the intermediate versions (prior to partitioning) will be run on a PC, the PFP host, and a single PFP target.

With regard to the uninitialized variables, we examined the GTRI code and came up with some reasonable initial values, which enabled us to run our executive program with the GTRI vehicle model. We tried running the simulation with an initially-stationary vehicle, as well as with a non-zero initial velocity, both with and without gravity. The output indicates that the model behaves correctly, at least from a qualitative standpoint. (The stationary vehicle falls toward earth, and the moving vehicle continues moving).

We then continued with a quantitative evaluation of the simulation's performance. We let the LEAP vehicle free-fall and observed that it fell at the correct rate. We also imparted an initial spin and verified that it remained constant. We then calculated the correct altitude and initial velocity to achieve geosynchronous orbit. When these values were used in the simulation, the vehicle behaved approximately as expected, showing only a slight degradation of orbit after several hundred seconds. By making the integration timestep larger, the degradation got worse, which can be attributed to numerical inaccuracies. We may investigate smaller timesteps and other integration algorithms, like RK4 (we are currently just using Euler, as in EXOSIM, but Brian uses RK4 and variable-step methods).

It was confirmed that it is possible to change most of the LEAP variables to single-precision, without significant effects on the program output. A temporary version was made in which the only variables left as double precision were:

-- Earth coordinate system variables, including initial values

-- All coordinate-transformation matrices

-- Gravity calculations.

The gravity calculations are done in double-precision partly because they are based on Earth coordinates and partly because the results can be very small. All intermediate versions of LEAP will be left with only double-precision real numbers for the near future, since the GTRI source uses double-precision. At some later time, we may choose to use more single-precision variables to accomodate the FPP and the crossbar.

Graphic display capability was added to the program. We can plot orbits on the PC screen, and we have run test cases of near-circular, elliptical, and parabolic trajectories. The graphics are

fairly simple, which may allow them to be ported to our graphics terminals on the PFP by simply substituting a few routines.

Brian Stevens subsequently made some changes in his continuous-time routine, "F". Most of these changes related to the coordinate systems, which had been redefined. Brian also cleaned up some of the COMMON blocks, implemented cross-product torque terms, altered his use of the quaternion, and eliminated some unnecessary variables.

Few changes were required to make this version of the main LEAP routines run with our executive program. We simply had to replace some non-standard DO loops, correct some double-precision constants, and modify the implementation of an initialization routine (to avoid using the ENTRY statement, which is not supported on the Intel compiler). Three trajectories were run to verify that the new version was consistent with the older one. All data is now output in a format compatible with Excel, for easy plotting of values. For the first time, LEAP was ported to the RMX host. It compiled and ran fine, producing values that were identical to those from the PC version.

The working relationship with GTRI was productive, and their simulation was both lean and modular. We will have to consider during the coming year whether to continue with this version of LEAP or perhaps to work with the Hughes version.

6. Appendix A: Environment file format

The "ENVIRONMENT" file contains information necessary for mapping symbolic names used by the PFP development tools to actual hardware. It is a text file, and each line contains either information about a hardware element (crossbar, sequencer, or target processor) or a comment (always with a "#" as the first character on the line). Example 1shows a full 32-processor configuration. Normally, the ENVIRONMENT file does not need to be altered by the programmer. It may be necessary to do so, however, if some processors are removed for service or if memory settings are changed.

The form of a non-comment line in the ENVIRONMENT file is:

<element name> = <base address>;<limit address>;<element type>;

where <element name> is the label used by other applications to refer to that element, <base address> is the starting memory address of the element in the host address space, <limit address> is the number of valid memory locations (in bytes), and <element type> is one of several valid element types. Currently the only element types supported are 80286, 80386, 29325, 29327, 0001, 0002, effe, and fffe. Four of these are processor types (80286, 80386, 29325, and 29327), two are for the "first" crossbar and sequencer (0001 and effe, respectively), and two are for the "second" crossbar and sequencer in a 64-processor system (0002 and fffe, repectively). All numeric fields are hexadecimal.

The <element name> field can be any 16-character string, as long as there are no repetitions. These names are used in the PROCESS.TXT and NETWORK.TXT files for each application, so they should usually not be changed from their default values (or else some applications will cease to run correctly).

The <base address> field is not actually a true physical address. Only the last six hex digits represent the memory address of each element. The first two hex digits are used to "turn on" the appropriate card cage, since there are at least four active card cages in a PFP, all mapped to the same address space but with no more than one enabled at any given time. This is done by issuing a particular I/O command to the address 8XX, where the X's are the first two digits. All of this is transparent to the programmer, so the eight-digit address can be viewed as a virtual address.

Note that the example is for the "second" half of a 64-processor system. The first half would contain processors p00 through p31.

Example 1: ENVIRONMENT.

```
# network 2 configuration
crossbar = 00040000;010000;0002;
sequencer = 00000000;010000;fffe;
# upper right bank configuration
p58 = 02100000;100000;80286;
p33 = 02200000;100000;80286;
p37 = 02300000;100000;80286;
p48 = 02400000;100000;80286;
p52 = 02500000;100000;80286;
p47 = 02600000;100000;80286;
p43 = 02700000;100000;80286;
p63 = 02800000;100000;80286;
p59 = 02900000;100000;80286;
p32 = 02a00000;100000;80286;
p36 = 02b00000;100000;80286;
# middle right bank configuration
p45 = 04100000;100000;80286;
p41 = 04200000;100000;80286;
p61 = 04300000;100000;80286;
p57 = 04400000;100000;80286;
p34 = 04500000;100000;80286;
p38 = 04600000;100000;80286;
p49 = 04700000;100000;80286;
p53 = 04800000;100000;80286;
p46 = 04900000;100000;80286;
p42 = 04a00000;100000;80286;
p62 = 04b00000;100000;80286;
# lower right bank configuration
p51 = 06100000;100000;80286;
p55 = 06200000;100000;80286;
p44 = 06300000;100000;80286;
p40 = 06400000;100000;80286;
p60 = 06500000;100000;80286;
p56 = 06600000;100000;80286;
p35 = 06700000;100000;80286;
p39 = 06800000;100000;80286;
p50 = 06900000;100000;80286;
p54 = 06a00000;100000;80286;
# <element name> = <base address>;<limit address>;<element type>;

.
```

7. Appendix B: vicld program source

FILE: vicld/Makefile

```
#
# Copyright 1991
# Georgia Institute of Technology
# Computer Engineering Research Laboratory
# Author:  Stephen R. Wachtel
#

default:        vicld

CC = cc -g
INCLUDE =
CFLAGS =

vicld:  vicld.o vicld_sym.o
        $(CC) -o vicld vicld.o vicld_sym.o

.SUFFIXES: .c .o
.c.o:
        $(CC) -c $(CFLAGS) $<

clean:
        rm -r vicld vicld.o vicld_sym.o
```

FILE: vicld/a.out.h

```
/*
 * Copyright 1991
 * Georgia Institute of Technology
 * Computer Engineering Research Laboratory
 */

#include "magic.h"

/*
        exec header format
*/
struct exec {
        unsigned long   a_magic;        /* magic number */
        unsigned long   a_text;         /* text segment size */
        unsigned long   a_data;         /* initialized data size*/
        unsigned long   a_bss;          /* uninitialized data size*/
        unsigned long   a_syms;         /* symbol table size*/
        unsigned long   a_entry;        /* entry point */
        unsigned long   a_trsize;       /* text relocation size*/
        unsigned long   a_drsize;       /* data relocation size*/
};

#define N_BADMAG(x) \
        (BAD_PFP_MAGIC((x).a_magic))

/*
        object file section offsets
*/

#define N_TXTOFF(x) \
        sizeof(struct exec)
#define N_DTAOFF(x) \
        (N_TXTOFF(x) + (x).a_text)
#define N_TRLOFF(x) \
        (N_DTAOFF(x) + (x).a_data)
#define N_DRLOFF(x) \
        (N_TRLOFF(x) + (x).a_trsize)
#define N_SYMOFF(x) \
        (N_DRLOFF(x) + (x).a_drsize)
#define N_STROFF(x) \
```

```
        (N_SYMOFF(x) + (x).a_syms)


/*
        relocation information format
*/
struct relocation_info
{
        long    r_address;      /* address which is relocated */
unsigned int    r_symbolnum:24,         /* local symbol ordinal */
                r_pcrel:1,      /* was relocated pc relative already */
                r_length:2,     /* 0=byte, 1=2 bytes, 2=4 bytes, 3=<invalid> */
                r_extern:1,     /* does not include value of sym referenced */
                :4;             /* unused */
};



/*
        symbol table entry format
*/
struct  nlist {
        union {
                char    *n_name;        /* for use when in-core */
                long    n_strx;         /* index into file string table */
        } n_un;
        unsigned char  n_type;          /* type flag (N_TEXT,..)  */
        char    n_other;                /* unused */
        short   n_hash;                 /* see <stab.h> */
        unsigned long  n_value;         /* value of symbol (or sdb offset) */
};

/*
        Simple values for n_type.
*/
#define N_UNDF  0x0             /* undefined */
#define N_ABS   0x2             /* absolute */
#define N_TEXT  0x4             /* text */
#define N_DATA  0x6             /* data */
#define N_BSS   0x8             /* bss */
#define N_COMM  0x12            /* common (internal to ld) */
#define N_FN    0x1f            /* file name symbol */
#define N_EXT   01              /* external bit, or'ed in */
#define N_TYPE  0x1e            /* mask for all the type bits */

/*
        Dbx entries have some of the N_STAB bits set.
        These are given in <stab.h>
*/
#define N_STAB  0xe0            /* if any of these bits set, a dbx symbol */


FILE: vicld/magic.h


/*
 * Copyright 1991
 * Georgia Institute of Technology
 * Computer Engineering Research Laboratory
 */


#define PFP_MAGIC_FPP_LOADABLE 0x50504603      /* "\003FPP" */
#define PFP_MAGIC_FPP_LINKABLE         0x70706603      /* "\003fpp" */
#define PFP_MAGIC_FPX_LOADABLE 0x58504603      /* "\003FPX" */
#define PFP_MAGIC_FPX_LINKABLE         0x78706603      /* "\003fpx" */
#define PFP_MAGIC_SEQ_LOADABLE         0x51455303      /* "\003SEQ" */
#define PFP_MAGIC_XBAR_LOADABLE        0x52415803      /* "\003XBR" */
#define PFP_MAGIC_286_LOADABLE         0x36383203      /* "\003286" */
#define PFP_MAGIC_286_KERNEL   0x4b383203      /* "\00328K" */
#define PFP_MAGIC_286_BOOTSTRAP        0x42383203      /* "\00328B" */
#define PFP_MAGIC_386_LOADABLE         0x36383303      /* "\003386" */
#define PFP_MAGIC_386_KERNEL   0x4b383303      /* "\00328K" */
#define PFP_MAGIC_386_BOOTSTRAP        0x42383303      /* "\00338B" */
#define PFP_MAGIC_386_COFF     0x0004014c      /* by inspection */

#define BAD_PFP_MAGIC(x) \
        ((x) != PFP_MAGIC_FPP_LOADABLE                          \
                && (x) != PFP_MAGIC_FPP_LINKABLE        \
                && (x) != PFP_MAGIC_FPX_LOADABLE        \
                && (x) != PFP_MAGIC_FPX_LINKABLE        \
```

```
                && (x) != PFP_MAGIC_SEQ_LOADABLE        \
                && (x) != PFP_MAGIC_XBAR_LOADABLE       \
                && (x) != PFP_MAGIC_286_LOADABLE        \
                && (x) != PFP_MAGIC_286_KERNEL                  \
                && (x) != PFP_MAGIC_286_BOOTSTRAP       \
                && (x) != PFP_MAGIC_386_LOADABLE        \
                && (x) != PFP_MAGIC_386_KERNEL                  \
                && (x) != PFP_MAGIC_386_BOOTSTRAP       \
                && (x) != PFP_MAGIC_386_COFF            \
        )

FILE: vicld/vicld.c


/*
 * Copyright 1991
 * Georgia Institute of Technology
 * Computer Engineering Research Laboratory
 * Author:  Stephen R. Wachtel
 */



#include <stdio.h>
#include <string.h>
#include "a.out.h"
#include "vicld.h"
#include "vicld_sym.h"



#define TEXT_SIZE 12
#define DATA_SIZE 4
#define BSS_SIZE  DATA_SIZE



int map = 0;
int number_error = 0;
int number_object = 0;



#define NUMBER_OBJECT 1024
OBJECT input_object[ NUMBER_OBJECT ];
OBJECT output_object;



int compare( entry1, entry2 )
RELOCATION *entry1;
RELOCATION *entry2;
{
        /* qsort compare */
        return( entry1->r_address - entry2->r_address );
} /* compare */



void read_object_header( object )
OBJECT *object;
{
        /* read header */
        fread( &object->header, sizeof( HEADER ), 1, object->file );

        /* check header */
        switch ( object->header.a_magic )
        {
                case PFP_MAGIC_FPP_LINKABLE:
                        output_object.header.a_magic = PFP_MAGIC_FPP_LOADABLE;
                        break;

                case PFP_MAGIC_FPX_LINKABLE:
                        output_object.header.a_magic = PFP_MAGIC_FPX_LOADABLE;
                        break;

                default:
                        fprintf( stderr, "ERROR: magic number '%s'\n", object->name );
                        exit( 1 );
        }
```

```
} /* read_object_header */


void read_object_string( object )
OBJECT *object;
{
        int length;
        STRING *string;

        /* read string table length */
        fseek( object->file, (long)N_STROFF( object->header ), 0 );
        fread( &length, sizeof( length ), 1, object->file );

        /* allocate string table */
        string = (STRING *)error_malloc( length );

        /* read string table */
        fseek( object->file, (long)N_STROFF( object->header ), 0 );
        fread( string, length, 1, object->file );

        /* save string table */
        object->string = string;
} /* read_object_string */


void read_object_symbol( object )
OBJECT *object;
{
        int length;
        SYMBOL *symbol;

        /* read symbol table length */
        length = object->header.a_syms;

        /* allocate symbol table */
        symbol = (SYMBOL *)error_malloc( length );

        /* read symbol table */
        fseek( object->file, (long)N_SYMOFF( object->header ), 0 );
        fread( symbol, length, 1, object->file );

        /* store symbol table */
        object->symbol = symbol;
} /* read_object_symbol */


void read_object_tr( object )
OBJECT *object;
{
        int length;
        RELOCATION *tr;

        /* read text relocation table length */
        length = object->header.a_trsize;

        /* allocate text relocation table */
        tr = (RELOCATION *)error_malloc( length );

        /* read text relocation table */
        fseek( object->file, (long)N_TRLOFF( object->header ), 0 );
        fread( tr, length, 1, object->file );

        /* sort text relocation table */
        qsort( tr, length / sizeof( RELOCATION ), sizeof( RELOCATION ), compare );

        /* store text relocation table */
        object->tr = tr;
} /* read_object_tr */


void read_object_dr( object )
OBJECT *object;
{
        int length;
        RELOCATION *dr;

        /* read data relocation table length */
```

```
        length = object->header.a_drsize;

        /* allocate data relocation table */
        dr = (RELOCATION *)error_malloc( length );

        /* read data relocation table */
        fseek( object->file, (long)N_DRLOFF( object->header ), 0 );
        fread( dr, length, 1, object->file );

        /* sort data relocation table */
        qsort( dr, length / sizeof( RELOCATION ), sizeof( RELOCATION ), compare );

        /* store data relocation table */
        object->dr = dr;
} /* read_object_dr */


void update_object_symbol( object )
OBJECT *object;
{
        int length;
        register int index;

        /* calculate symbol table length */
        length = object->header.a_syms / sizeof( SYMBOL );

        /* replace symbol index with symbol name */
        for ( index = 0; index != length; index++ )
        {
                object->symbol[ index ].n_un.n_name = &object->string[ object->symbol[
index ].n_un.n_strx ];
        }
} /* update_object_symbol */


void read_input_object( )
{
        register int object_number;

        for ( object_number = 0; object_number != number_object; object_number++ )
        {
                if ( ( input_object[ object_number ].file = fopen( input_object[
object_number ].name, "r" ) ) == NULL )
                {
                        fprintf( stderr, "ERROR: unable to open '%s'\n", input_object[
object_number ].name );
                        exit( 1 );
                }

                read_object_header( &input_object[ object_number ] );

                read_object_string( &input_object[ object_number ] );

                read_object_symbol( &input_object[ object_number ] );

                read_object_tr( &input_object[ object_number ] );

                read_object_dr( &input_object[ object_number ] );

                update_object_symbol( &input_object[ object_number ] );

                fclose( input_object[ object_number ].file );
        }
} /* read_input_object */


void print_object_header( object )
OBJECT *object;
{
        fprintf( stdout, "HEADER:\n" );

        switch ( object->header.a_magic )
        {
                case PFP_MAGIC_FPP_LINKABLE:
                        fprintf( stdout, "fpp object\n" );
                        break;

                case PFP_MAGIC_FPX_LINKABLE:
```

```
                fprintf( stdout, "fpx object\n" );
                break;

        default:
                fprintf( stderr, "ERROR: magic number '%s'\n", object->name );
                exit( 1 );
        }

        fprintf( stdout, "text size = %d\n", object->header.a_text );
        fprintf( stdout, "data size = %d\n", object->header.a_data );
        fprintf( stdout, "bss  size = %d\n", object->header.a_bss  );

        fprintf( stdout, "entry address = %d\n",
        object->header.a_entry );
        fprintf( stdout, "text relocation table length = %d\n",
        object->header.a_trsize / sizeof( RELOCATION ) );
        fprintf( stdout, "data relocation table length = %d\n",
        object->header.a_drsize / sizeof( RELOCATION ) );
        fprintf( stdout, "symbol table length = %d\n",
        object->header.a_syms / sizeof( SYMBOL ) );

        fprintf( stdout, "\n" );
} /* print_object_header */


void print_object_symbol( object )
OBJECT *object;
{
        register int index;
        int length;
        SYMBOL *symbol;

        fprintf( stdout, "SYMBOL TABLE:\n" );

        /* calculate symbol table length */
        length = object->header.a_syms / sizeof( SYMBOL );

        /* print symbol table record */
        for ( index = 0; index != length; index++ )
        {
                symbol = &object->symbol[ index ];

                fprintf( stdout, "%s %d %d %d %d\n",
                symbol->n_un.n_name. symbol->n_type, symbol->n_other, symbol->n_hash,
symbol->n_value );
        }

        fprintf( stdout, "\n" );
} /* print_object_symbol */


void print_object_tr( object )
OBJECT *object;
{
        register int index;
        int length;
        RELOCATION *tr;

        fprintf( stdout, "TEXT RELOCATION TABLE:\n" );

        /* calculate text relocation table length */
        length = object->header.a_trsize / sizeof( RELOCATION );

        /* print text relocation table record */
        for ( index = 0; index != length; index++ )
        {
                tr = &object->tr[ index ];

                fprintf( stdout, "%d %d %d %d %d\n",
                tr->r_address, tr->r_symbolnum, tr->r_pcrel, tr->r_length, tr->r_extern );
        }

        fprintf( stdout, "\n" );
} /* print_object_tr */


void print_object_dr( object )
OBJECT *object;
```

```
{
        register int index;
        int length;
        RELOCATION *dr;

        fprintf( stdout, "DATA RELOCATION TABLE:\n" );

        /* calculate data relocation table length */
        length = object->header.a_drsize / sizeof( RELOCATION );

        /* print data relocation table record */
        for ( index = 0; index != length; index++ )
        {
                dr = &object->dr[ index ];

                fprintf( stdout, "%d %d %d %d %d\n",
                dr->r_address, dr->r_symbolnum, dr->r_pcrel, dr->r_length, dr->r_extern );
        }

        fprintf( stdout, "\n" );
} /* print_object_dr */


void print_input_object( )
{
        register int object_number;

        for ( object_number = 0; object_number != number_object; object_number++ )
        {
                fprintf( stdout, "%s (%s):\n", input_object[ object_number ].name,
                ( input_object[ object_number ].required ? "required" : "not required" ) );

                print_object_header( &input_object[ object_number ] );

                print_object_symbol( &input_object[ object_number ] );

                print_object_tr( &input_object[ object_number ] );

                print_object_dr( &input_object[ object_number ] );

                fprintf( stdout, "\n" );
        }
} /* print_input_object */


void add_public_symbol( )
{
        register int object_number;
        register int index;
        int length;
        SYMBOL *symbol;
        STRING *string;
        LIST *list;

        for ( object_number = 0; object_number != number_object; object_number++ )
        {
                /* calculate symbol table length */
                length = input_object[ object_number ].header.a_syms / sizeof( SYMBOL );

                /* search symbol table for public symbol */
                for ( index = 0; index != length; index++ )
                {
                        symbol = &input_object[ object_number ].symbol[ index ];

                        if ( ( symbol->n_type & N_EXT ) != N_EXT )
                        {
                                string = symbol->n_un.n_name;

                                if ( find_symbol( string ) == NULL )
                                {
                                        list = add_symbol( string );
                                        list->n_type = symbol->n_type;
                                        list->n_other = symbol->n_other;
                                        list->n_hash = symbol->n_hash;
                                        list->n_value = symbol->n_value;
                                        list->object_number = object_number;
                                }
                                else
                                        fprintf( stdout, "WARNING: multiple declaration
```

```
'%s'\n", string );
                        }
                }
        }
} /* add_public_symbol */


void resolve_external_symbol( object_number )
int object_number;
{
        /* recursive */
        int index;
        int length;
        RELOCATION *tr;
        RELOCATION *dr;
        STRING *string;
        LIST *list;

        if ( input_object[ object_number ].required )
                return;

        input_object[ object_number ].required = 1;

        /* calculate text relocation table length */
        length = input_object[ object_number ].header.a_trsize / sizeof( RELOCATION );

        /* search text relocation table for external symbol */
        for ( index = 0; index != length; index++ )
        {
                tr = &input_object[ object_number ].tr[ index ];

                if ( tr->r_extern )
                {
                        string = input_object[ object_number ].symbol[ tr->r_symbolnum
].n_un.n_name;

                        list = find_symbol( string );

                        if ( list != NULL )
                                resolve_external_symbol( list->object_number );
                        else
                        {
                                fprintf( stderr, "WARNING: unresolved reference '%s'\n",
string );

                                list = add_symbol( string );
                                number_error = 1;
                        }
                }
        }

        /* calculate data relocation table length */
        length = input_object[ object_number ].header.a_drsize / sizeof( RELOCATION );

        /* search data relocation table for external symbol */
        for ( index = 0; index != length; index++ )
        {
                dr = &input_object[ object_number ].dr[ index ];

                if ( dr->r_extern )
                {
                        string = input_object[ object_number ].symbol[ dr->r_symbolnum
].n_un.n_name;

                        list = find_symbol( string );

                        if ( list != NULL )
                                resolve_external_symbol( list->object_number );
                        else
                        {
                                fprintf( stderr, "WARNING: unresolved reference '%s'\n",
string );

                                list = add_symbol( string );
                                number_error = 1;
                        }
                }
        }
} /* resolve_external_symbol */
```

```
void write_output_header( )
{
        register int object_number;

        output_object.header.a_text = 0;
        output_object.header.a_data = 0;
        output_object.header.a_bss  = 0;

        for ( object_number = 0; object_number != number_object; object_number++ )
        {
                if ( !input_object[ object_number ].required )
                        continue;

                output_object.header.a_text += input_object[ object_number ].header.a_text;
                output_object.header.a_data += input_object[ object_number ].header.a_data;
                output_object.header.a_bss  += input_object[ object_number ].header.a_bss ;
        }

        output_object.header.a_entry = N_TXTOFF( output_object.header );
        output_object.header.a_trsize = 0;
        output_object.header.a_drsize = 0;
        output_object.header.a_syms = 0;

        fwrite( &output_object.header, sizeof( output_object.header ), 1,
output_object.file );
} /* write_output_header */


void print_map( )
{
        register int object_number;

        fprintf( stdout, "MAP:\n" );

        for ( object_number = 0; object_number != number_object; object_number++ )
        {
                if ( !input_object[ object_number ].required )
                        continue;

                fprintf( stdout, "%-64s ", input_object[ object_number ].name );
                fprintf( stdout, "%6d ", input_object[ object_number ].header.a_text );
                fprintf( stdout, "%6d\n", input_object[ object_number ].header.a_data +
input_object[ object_number ].header.a_bss );
        }

        fprintf( stdout, "%-64s ", "TOTAL");
        fprintf( stdout, "%6d ", output_object.header.a_text );
        fprintf( stdout, "%6d\n", output_object.header.a_data + output_object.header.a_bss
);
} /* print_map */


void update_public_symbol( )
{
        register int object_number;
        register int index;
        int length;
        SYMBOL *symbol;
        LIST *list;

        output_object.text = 0;
        output_object.data = 0;
        output_object.bss  = ( output_object.header.a_data / DATA_SIZE );

        for ( object_number = 0; object_number != number_object; object_number++ )
        {
                if ( !input_object[ object_number ].required )
                        continue;

                input_object[ object_number ].text = output_object.text;
                input_object[ object_number ].data = output_object.data;
                input_object[ object_number ].bss  = output_object.bss ;

                /* calculate symbol table length */
                length = input_object[ object_number ].header.a_syms / sizeof( SYMBOL );

                /* search symbol table for public symbol */
```

```
                for ( index = 0; index != length; index++ )
                {
                        symbol = &input_object[ object_number ].symbol[ index ];

                        if ( ( symbol->n_type & N_EXT ) != N_EXT )
                        {
                                list = find_symbol( symbol->n_un.n_name );

                                switch ( list->n_type & N_TYPE )
                                {
                                        case N_TEXT:
                                                list->n_value += input_object[ object_number
].text;
                                                break;

                                        case N_DATA:
                                                list->n_value += input_object[ object_number
].data;
                                                break;

                                        case N_BSS:
                                                list->n_value += input_object[ object_number
].bss ;
                                                break;
                                }
                        }
                }

                output_object.text += ( input_object[ object_number ].header.a_text /
TEXT_SIZE );
                output_object.data += ( input_object[ object_number ].header.a_data /
DATA_SIZE );
                output_object.bss  += ( input_object[ object_number ].header.a_bss  /
BSS_SIZE  );
        }
} /* update_public_symbol */


void write_object_text( object )
OBJECT *object;
{
        register int index;
        int length;
        int address;
        RELOCATION *tr;
        unsigned short text;
        LIST *list;

        fseek( object->file, (long)N_TXTOFF( object->header ), 0 );
        address = 0;

        /* calculate text relocation table length */
        length = object->header.a_trsize / sizeof( RELOCATION );

        /* relocate text relocation table record */
        for ( index = 0; index != length; index++ )
        {
                tr = &object->tr[ index ];

                /* skip text */
                while ( address < tr->r_address )
                {
                        fread( &text, sizeof( text ), 1, object->file );
                        fwrite( &text, sizeof( text ), 1, output_object.file );
                        address += sizeof( text );
                }

                /* relocate text */
                fread( &text, sizeof( text ), 1, object->file );

                if ( tr->r_extern )
                {
                        list = find_symbol( object->symbol[ tr->r_symbolnum ].n_un.n_name
);

                        if ( list != NULL )
                                text += list->n_value;
                }
                else
                {
```

```
                        switch ( tr->r_symbolnum )
                        {
                                case N_TEXT:
                                        text += object->text;
                                        break;

                                case N_DATA:
                                        text += object->data;
                                        break;

                                case N_BSS:
                                        text += object->bss ;
                                        break;
                        }
                }

                fwrite( &text, sizeof( text ), 1, output_object.file );
                address += sizeof( text );
        }

        /* copy text */
        while ( address < object->header.a_text )
        {
                fread( &text, sizeof( text ), 1, object->file );
                fwrite( &text, sizeof( text ), 1, output_object.file );
                address += sizeof( text );
        }
} /* write_object_text */


void write_object_data( object )
OBJECT *object;
{
        register int index;
        int length;
        int address;
        RELOCATION *dr;
        unsigned long data;
        LIST *list;

        fseek( object->file, (long)N_DTAOFF( object->header ), 0 );
        address = 0;

        /* calculate data relocation table length */
        length = object->header.a_drsize / sizeof( RELOCATION );

        /* relocate data relocation table record */
        for ( index = 0; index != length; index++ )
        {
                dr = &object->dr[ index ];

                /* skip data */
                while ( address < dr->r_address )
                {
                        fread( &data, sizeof( data ), 1, object->file );
                        fwrite( &data, sizeof( data ), 1, output_object.file );
                        address += sizeof( data );
                }

                /* relocate data */
                fread( &data, sizeof( data ), 1, object->file );

                if ( dr->r_extern )
                {
                        list = find_symbol( object->symbol[ dr->r_symbolnum ].n_un.n_name
);

                        if ( list != NULL )
                                data += list->n_value;
                }
                else
                {
                        switch ( dr->r_symbolnum )
                        {
                                case N_TEXT:
                                        data += object->text;
                                        break;

                                case N_DATA:
                                        data += object->data;
```

```
                                        break;

                                case N_BSS:
                                        data += object->bss ;
                                        break;
                        }
                }

                fwrite( &data, sizeof( data ), 1, output_object.file );
                address += sizeof( data );
        }

        /* copy data */
        while ( address < object->header.a_data )
        {
                fread( &data, sizeof( data ), 1, object->file );
                fwrite( &data, sizeof( data ), 1, output_object.file );
                address += sizeof( data );
        }
} /* write_object_data */


void write_output_object( )
{
        register int object_number;

        /* relocate text */
        fseek( output_object.file, (long)N_TXTOFF( output_object.header ), 0 );

        for ( object_number = 0; object_number != number_object; object_number++ )
        {
                if ( !input_object[ object_number ].required )
                        continue;

                if ( ( input_object[ object_number ].file = fopen( input_object[
object_number ].name, "r" ) ) == NULL )
                {
                        fprintf( stderr, "ERROR: unable to open '%s'\n", input_object[
object_number ].name );
                        exit( 1 );
                }

                write_object_text( &input_object[ object_number ] );

                fclose( input_object[ object_number ].file );
        }

        /* relocate data */
        fseek( output_object.file, (long)N_DTAOFF( output_object.header ), 0 );

        for ( object_number = 0; object_number != number_object; object_number++ )
        {
                if ( !input_object[ object_number ].required )
                        continue;

                if ( ( input_object[ object_number ].file = fopen( input_object[
object_number ].name, "r" ) ) == NULL )
                {
                        fprintf( stderr, "ERROR: unable to open '%s'\n", input_object[
object_number ].name );
                        exit( 1 );
                }

                write_object_data( &input_object[ object_number ] );

                fclose( input_object[ object_number ].file );
        }
} /* write_output_object */


#define PROGRAM argument[ 0 ]
#define ARGUMENT argument[ 1 + argument_number ]


int main( number_argument, argument )
int number_argument;
char *argument[ ];
{
```

```
        register int argument_number;

        if ( --number_argument == 0 )
        {
                fprintf( stderr, "usage: %s <object> [<object> ...]\n", PROGRAM );
                exit( 1 );
        }
        fprintf( stderr, "FPP/FPX LOADER version: 1.0 07/10/91\n" );

        initialize_symbol_table( );

        for ( argument_number = 0; argument_number != number_argument; argument_number++ )
        {
                if ( strcmp( ARGUMENT, "-map" ) == 0 )
                {
                        map = 1;
                        continue;
                }

                input_object[ number_object ].name = ARGUMENT;
                input_object[ number_object ].required = 0;

                if ( ++number_object == NUMBER_OBJECT )
                {
                        fprintf( stderr, "ERROR: number_object == %d\n", NUMBER_OBJECT );
                        exit( 1 );
                }
        }

        read_input_object( );

        add_public_symbol( );

        resolve_external_symbol( 0 );

#ifdef DEBUG
        print_input_object( );
#endif

        output_object.name = strdup( "a.out" );
        output_object.required = 0;

        if ( ( output_object.file = fopen( output_object.name, "w+" ) ) == NULL )
        {
                fprintf( stderr, "ERROR: unable to open '%s'\n", output_object.name );
                exit( 1 );
        }

        write_output_header( );

        if ( map )
                print_map( );

        update_public_symbol( );

#ifdef DEBUG
        print_symbol_table( );
#endif

        write_output_object( );

        fclose( output_object.file );

        exit( number_error );
} /* main */


FILE: vicld/vicld.h


/*
 * Copyright 1991
 * Georgia Institute of Technology
 * Computer Engineering Research Laboratory
 * Author:  Stephen R. Wachtel
 */



#define HEADER struct exec
#define STRING char
```

```
#define SYMBOL struct nlist
#define RELOCATION struct relocation_info


#define OBJECT struct object_type
OBJECT
{
        char *name;
        FILE *file;
        HEADER header;
        STRING *string;
        SYMBOL *symbol;
        RELOCATION *tr;
        RELOCATION *dr;
        int text;
        int data;
        int bss;
        int required;
};


int compare( /* RELOCATION *entry1, RELOCATION *entry2 */ );
void read_object_header( /* OBJECT *object */ );
void read_object_string( /* OBJECT *object */ );
void read_object_symbol( /* OBJECT *object */ );
void update_object_symbol( /* OBJECT *object */ );
void read_object_tr( /* OBJECT *object */ );
void read_object_dr( /* OBJECT *object */ );
void read_input_object( /* void */ );
void print_object_header( /* OBJECT *object */ );
void print_object_symbol( /* OBJECT *object */ );
void print_object_tr( /* OBJECT *object */ );
void print_object_dr( /* OBJECT *object */ );
void print_input_object( /* void */ );
void add_public_symbol( /* void */ );
void resolve_external_symbol( /* int object_number */ );
void write_output_header( /* void */ );
void print_map( /* void */ );
void update_public_symbol( /* void */ );
void write_object_text( /* OBJECT *object */ );
void write_object_data( /* OBJECT *object */ );
void write_output_object( );
int main( /* int number_argument, char *argument[ ] */ );


FILE: vicld/vicld_sym.c


/*
 * Copyright 1991
 * Georgia Institute of Technology
 * Computer Engineering Research Laboratory
 */


/*
*       symbol --- symbol table and mapping routines
*/

#include <stdio.h>
#include <string.h>
#include "vicld_sym.h"

/*
*       MAXHASH --- determines the hash table width.
*       symtab  --- the symbol table structure
*/

#define MAXHASH        311
#define MAXESYMS 50

static LIST * symtab[MAXHASH];
/*
*       external routine declarations
*/

char *malloc();

/*
```

```
*       error_malloc --- error checking malloc routine
*/

char *
error_malloc(size)
unsigned int size;
{
        char *p;


        if (!(p = malloc(size)))
        {
                fprintf(stderr,"no more dynamic storage - aborting\n");
                exit(1);
        }

        return(p);
} /* error_malloc */


/*
*       hash --- scramble a name (hopefully) uniformly to fit in a table
*/

static unsigned int
hash(name)
register char *name;
{
        register unsigned int h = 0;
        while (*name)
        {
                h <<= 4;
                h ^= *name++;
        }

        return(h % MAXHASH);
}
/*
*       add_symbol --- enter a name into the symbol table
*/

LIST *
add_symbol(name)
char *name;
{
        register LIST * p;
        unsigned int h;

/*
*       create an entry and insert it at the front of the table
*/

        h = hash(name);

        p = (LIST *) error_malloc(sizeof(LIST));
        p->n_name = strdup(name);
        p->n_type = 0;
        p->n_other = 0;
        p->n_hash = 0;
        p->n_value = 0;
        p->object_number = 0;
        p->next = symtab[h];

        symtab[h] = p;
        return(p);
}
/*
*       find_symbol --- lookup a symbol in the symbol table
*
*               find_symbol scans the symbol table and returns a pointer to
*               the symbol table entry
*/

LIST *
find_symbol(name)
char *name;
{
        register LIST * p;
        unsigned int h;
        h = hash(name);
        for (p = symtab[h]; p != 0; p = p->next)
```

```
                if (strcmp(p->n_name, name) == 0)
                        break;

        return(p);
}

void initialize_symbol_table()
{
        bzero(symtab, MAXHASH * sizeof(LIST *));
} /* initialize_symbol_table */


void print_symbol_table()
{
        unsigned int h;
        register LIST *p;

        fprintf( stdout, "SYMBOL TABLE:\n" );

        for ( h = 0; h != MAXHASH; h++ )
        {
                for ( p = symtab[ h ]; p != 0; p = p->next )
                {
                        fprintf( stdout, "%s %d %d %d %d\n",
                        p->n_name, p->n_type, p->n_other, p->n_hash, p->n_value );
                }
        }

        fprintf( stdout, "\n" );
} /* print_symbol_table */
```

FILE: vicld/vicld_sym.h

```
/*
 * Copyright 1991
 * Georgia Institute of Technology
 * Computer Engineering Research Laboratory
 */


/* see struct nlist in a.out.h for explanantion of structure */


#define LIST struct list_type
LIST
{
        char *n_name;
        unsigned char n_type;
        char n_other;
        short n_hash;
        unsigned long n_value;
        int object_number;
        LIST *next;
};

char *error_malloc(/* unsigned int */);
LIST *add_symbol(/* char * */);
LIST *find_symbol(/* char * */);
void initialize_symbol_table(/* void */);
void print_symbol_table(/* void */);
```

8. Appendix C: loadfpp program source

FILE: loadfpp/Makefile

```
#
# Copyright 1991
# Georgia Institute of Technology
# Computer Engineering Research Laboratory
# Author:  Stephen R. Wachtel
#

cflags = large optimize(3) debug \
        searchinclude( :LIB:ic286/,:PFP:include/ )

loadfpp:        loadfpp.obj
        submit :PFP:csd/cbndl( loadfpp, loadfpp.obj, debug )

loadfpp.obj:    loadfpp.c
        ic286 loadfpp.c $(cflags)

clean:
        delete loadfpp,*.lst,*.obj,*.mp?
```

FILE: loadfpp/fpp.h

```
/*
 * Copyright 1991
 * Georgia Institute of Technology
 * Computer Engineering Research Laboratory
 * Author:  Stephen R. Wachtel
 */

FPP_CODE
{
        /* word 0 */
        unsigned        s_index_register:4;
        unsigned        r_index_register:4;
        unsigned        f_index_register:4;
        unsigned        s_index_flag:1;
        unsigned        r_index_flag:1;
        unsigned        f_index_flag:1;
        unsigned        msw_selector:1;

        /* word 1 */
        unsigned        s_address:16;

        /* word 2 */
        unsigned        r_address:16;

        /* word 3 */
        unsigned        f_address:16;

        /* word 4 */
        unsigned        mc325_opcode:3;
        unsigned        f_to_s_flag:1;
        unsigned        f_to_r_flag:1;
        unsigned        f_flag:1;
        unsigned        read_opcode:2;
        unsigned        branch_selector:1;
        unsigned        write_opcode:3;
        unsigned        branch_opcode:4;

        /* word 5 */
        unsigned        branch_address:12;
        unsigned        am2910_opcode:4;
};

#define FPP_DATA struct fpp_data_type
FPP_DATA
{
        /* word 0 */
        unsigned short lsw;
```

```
        /* word 1 */
        unsigned short msw;
};


FILE: loadfpp/loadfpp.c


/*
 * Copyright 1991
 * Georgia Institute of Technology
 * Computer Engineering Research Laboratory
 * Author:  Stephen R. Wachtel
 */


#include <stdio.h>
#include <stdlib.h>
#include <host.h>


#include "a.out.h"
#include "fpp.h"


#define DATA_PORT 0x0c000
#define STATUS_PORT 0x0e000


char *value;
unsigned long base;
unsigned long limit;
unsigned long type;


HEADER header;
unsigned short buffer[6][4096];


FPP_CODE instruction;
unsigned short program_counter;


void stop_processor( void )
{
        unsigned short status;

        status = 0;
        poke( base + STATUS_PORT, &status, sizeof( status ) );
        peek( base + STATUS_PORT, &status, sizeof( status ) );
        if ( ( status & 4 ) != 4 )
        {
                fprintf( stderr, "ERROR: unable to stop the processor\n" );
                exit( -1 );
        }
} /* stop_processor */


void start_processor( void )
{
        unsigned short status;

        status = 1;
        poke( base + STATUS_PORT, &status, sizeof( status ) );
        peek( base + STATUS_PORT, &status, sizeof( status ) );
        if ( ( status & 4 ) == 4 )
        {
                fprintf( stderr, "ERROR: unable to start the processor\n" );
                exit( -1 );
        }
} /* start_processor */


void send_data( unsigned short *buffer )
{
        unsigned short count;
        unsigned short status;

        for ( count = 0; count != 1024; count++ )
        {
                peek( base + STATUS_PORT, &status, sizeof( status ) );
```

```
                if ( ( status & 2 ) == 2 )
                {
                        poke( base + DATA_PORT, buffer, sizeof( *buffer ) );
                        return;
                }
        }

        fprintf( stderr, "ERROR: unable to send data\n" );
        exit( -1 );
} /* send_data */


void receive_data( unsigned short *buffer )
{
        unsigned short count;
        unsigned short status;

        for ( count = 0; count != 1024; count++ )
        {
                peek( base + STATUS_PORT, &status, sizeof( status ) );
                if ( ( status & 1 ) == 1 )
                {
                        peek( base + DATA_PORT, buffer, sizeof( *buffer ) );
                        return;
                }
        }

    fprintf( stderr, "ERROR: unable to receive data\n" );
        exit( -1 );
} /* receive_data */


void reset_instruction( void )
{
        /* word 5 */
        instruction.am2910_opcode = 14;
        instruction.branch_address = 0;

        /* word 4 */
        instruction.branch_opcode = 0;
        instruction.write_opcode = 0;
        instruction.branch_selector = 0;
        instruction.read_opcode = 0;
        instruction.f_flag = 0;
        instruction.f_to_r_flag = 0;
        instruction.f_to_s_flag = 0;
        instruction.mc325_opcode = 0;

        /* word 3 */
        instruction.f_address = 0;

        /* word 2 */
        instruction.r_address = 1;

        /* word 1 */
        instruction.s_address = 1;

        /* word 0 */
        instruction.msw_selector = 0;
        instruction.f_index_flag = 0;
        instruction.r_index_flag = 0;
        instruction.s_index_flag = 0;
        instruction.f_index_register = 0;
        instruction.r_index_register = 0;
        instruction.s_index_register = 0;
} /* reset_instruction */


void load_instruction( void )
{
        unsigned short index;

        if ( program_counter == 4096 )
        {
                fprintf( stderr, "ERROR: number instruction > 4096\n" );
                exit( -1 );
        }

        for ( index = 0; index != 6; index++ )
        {
                buffer[ index ][ program_counter ] = ( (unsigned short *)&instruction )[
```

```
index ];
        }

        program_counter++;
} /* load_instruction */


void generate_receive( unsigned short address )
{
        unsigned short count = 2;

        reset_instruction( );

        instruction.f_address = address;
        if ( address == 1 )
        {
                instruction.s_address = 0;
                instruction.r_address = 0;
        }

        while ( count != 0 )
        {
                instruction.msw_selector = --count;

                instruction.am2910_opcode = 14;
                instruction.branch_address = 0;
                instruction.branch_opcode = 0;
                instruction.write_opcode = 0;
                load_instruction( );

                instruction.am2910_opcode = 3;
                instruction.branch_address = program_counter;
                instruction.branch_opcode = 2;
                instruction.write_opcode = 2;
                load_instruction( );
        }
} /* generate_receive */


void generate_send( unsigned short address )
{
        unsigned short count = 2;

        reset_instruction( );

        instruction.s_address = address;
        instruction.r_address = address;
        if ( address == 0 )
        {
                instruction.f_address = 1;
        }

        while ( count != 0 )
        {
                instruction.msw_selector = --count;

                instruction.am2910_opcode = 14;
                instruction.branch_address = 0;
                instruction.branch_opcode = 0;
                instruction.read_opcode = 0;
                load_instruction( );

                instruction.am2910_opcode = 3;
                instruction.branch_address = program_counter;
                instruction.branch_opcode = 3;
                instruction.read_opcode = 2;
                load_instruction( );
        }
} /* generate_send */


void load_boot( char *path )
{
        FILE *file;
        unsigned short length;
        unsigned short offset;

        stop_processor( );

        if ( ( file = fopen( path, "rb" ) ) == (FILE *)NULL )
        {
```

```
                fprintf( stderr, "ERROR: unable to open for read '%s'\n", path );
                exit( -1 );
        }

        if ( fread( &header, sizeof( header ), 1, file ) != 1 )
        {
                fprintf( stderr, "ERROR: unable to read header\n" );
                exit( -1 );
        }

        program_counter = 0;

        reset_instruction( );
        load_instruction( );

        length = header.a_data / sizeof( FPP_DATA );

        for ( offset = 0; offset != length; offset++ )
        {
                generate_receive( offset );
#ifdef DEBUG
                generate_send( offset );
#endif
        }

        reset_instruction( );
        instruction.am2910_opcode = 3;
        instruction.branch_opcode = 12;
        instruction.branch_address = program_counter;
        load_instruction( );

        poke( base, buffer, sizeof( buffer ) );

        fclose( file );
} /* load_boot */


void load_data( char *path )
{
        FILE *file;
        unsigned short length;
        unsigned short offset;
        FPP_DATA w_buffer;
        FPP_DATA r_buffer;

        start_processor( );

        if ( ( file = fopen( path, "rb" ) ) == (FILE *)NULL )
        {
                fprintf( stderr, "ERROR: unable to open for read '%s'\n", path );
                exit( -1 );
        }

        if ( fread( &header, sizeof( header ), 1, file ) != 1 )
        {
                fprintf( stderr, "ERROR: unable to read header\n" );
                exit( -1 );
        }

        length = header.a_data / sizeof( FPP_DATA );
        fseek( file, N_DTAOFF( header ), 0 );

        for ( offset = 0; offset != length; offset++ )
        {
                if ( fread( &w_buffer, sizeof( FPP_DATA ), 1, file ) != 1 )
                {
                        fprintf( stderr, "ERROR: unable to read data\n" );
                        exit( -1 );
                }

                send_data( &w_buffer.msw );
                send_data( &w_buffer.lsw );
#ifdef DEBUG
                receive_data( &r_buffer.msw );
                receive_data( &r_buffer.lsw );

                if ( ( w_buffer.msw != r_buffer.msw ) || ( w_buffer.lsw != r_buffer.lsw ) )
                {
                        fprintf( stderr, "ERROR: data(%u), w=%04x%04x, r=%04x%04x\n",
                        offset, w_buffer.msw, w_buffer.lsw, r_buffer.msw, r_buffer.lsw );
                        exit( -1 );
```

```
                }
#endif
        }

        fclose( file );
} /* load_boot */


void load_code( char *path )
{
        FILE *file;
        unsigned short length;
        unsigned short offset;

        stop_processor( );

        if ( ( file = fopen( path, "rb" ) ) == (FILE *)NULL )
        {
                fprintf( stderr, "ERROR: unable to open for read '%s'\n", path );
                exit( -1 );
        }

        if ( fread( &header, sizeof( header ), 1, file ) != 1 )
        {
                fprintf( stderr, "ERROR: unable to read header\n" );
                exit( -1 );
        }

        program_counter = 0;

        length = header.a_text / sizeof( FPP_CODE );
        fseek( file, N_TXTOFF( header ), 0 );

        for ( offset = 0; offset != length; offset++ )
        {
                if ( fread( &instruction, sizeof( FPP_CODE ), 1, file ) != 1 )
                {
                        fprintf( stderr, "ERROR: unable to read code\n" );
                        exit( -1 );
                }

                load_instruction( );
        }

        poke( base, buffer, sizeof( buffer ) );

        fclose( file );
} /* load_code */


#define PROGRAM argument[ 0 ]
#define ARGUMENT argument[ argument_number ]


void main( int number_argument, char *argument[ ] )
{
        int argument_number = 0;
        char name[256];
        char path[256];

        initialize_environment( ":HOME:ENVIRONMENT" );

        if ( --number_argument == 0 )
        {
                fprintf( stderr, "usage: %s <name>=<path>...<name>=<path>\n", PROGRAM );
                exit( 0 );
        }

        while ( argument_number++ != number_argument )
        {
                if ( sscanf( ARGUMENT, "%[^=]=%s", name, path ) != 2 )
                {
                        fprintf( stderr, "ERROR: unable to parse argument '%s'\n", ARGUMENT
);
                        exit( -1 );
                }

                if ( ( value = getenv( name ) ) == NULL )
                {
                        fprintf( stdout, "ERROR: '%s' not found in environment\n", name );
                        exit( -1 );
```

```
                }

                if ( sscanf( value, "%lx;%lx;%lx;", &base, &limit, &type ) != 3 )
                {
                        fprintf( stdout, "ERROR: unable to parse '%s = %s'\n", name, value
);
                        exit( -1 );
                }

                fprintf( stdout, "loading %s\n", name );
                load_boot( path );
                load_data( path );
                load_code( path );

                fprintf( stdout, "starting %s\n", name );
                start_processor( );
        }

        exit( 0 );
} /* main */
```